Revisión Sistemática: Clean Architecture en ASP.NET Core 9 y Blazor

Brayan Pilla, Carlos Ramos

May 16, 2025

Abstract

La Clean Architecture, propuesta por Robert C. Martin, se ha vuelto cada vez más popular como una forma efectiva de crear aplicaciones que cuenten con la facilidad de ser mantenibles y escabalables con el tiempo. Su idea principal es mantener bien separada la lógica del negocio de los elementos técnicos, como la base de datos o la interfaz de usuario, lo que ayuda a que el software sea más estable y adaptable a cambios. Esta investigación revisa trabajos recientes sobre cómo se está usando Clean Architecture en proyectos desarrollados con ASP.NET Core 9 y Blazor, dos tecnologías importantes dentro del entorno .NET. Se explican sus principios básicos, las ventajas que ofrece, cómo se organiza estructuralmente y los patrones más comunes para aplicarla. También se analiza un caso práctico donde se implementa esta arquitectura en una solución moderna usando Blazor, mostrando de manera concreta cómo puede aplicarse en proyectos reales. En resumen, el estudio brinda una mirada actual sobre cómo este enfoque ayuda a crear aplicaciones web más flexibles y listas para evolucionar.

1 Introduction

En los últimos años, el desarrollo de software se ha enfrentado al reto de construir sistemas que no solo funcionen bien, sino que también sean fáciles de mantener, ampliar y ajustar con el paso del tiempo. Para responder a esta necesidad, han surgido varias propuestas de arquitectura pensadas para mejorar tanto la calidad del diseño como la vida útil del código. Entre todas ellas, la Clean Architecture ha destacado como una de las más completas y efectivas para enfrentar estos desafíos.

Propuesta por Robert C. Martin, la Clean Architecture plantea una separación estricta entre la lógica de negocio y los elementos de infraestructura, como bases de datos, interfaces gráficas o servicios externos. Este modelo busca mantener el núcleo del sistema, sus reglas y procesos fundamentales, aislado de los detalles técnicos que pueden cambiar con el tiempo, lo cual facilita su evolución continua, mejora la comprensión del código y reduce el impacto de las modificaciones [1].

El presente trabajo examina investigaciones recientes sobre cómo se ha estado implementando Clean Architecture en el uso de ASP.NET Core 9 y Blazor, dos herramientas clave del ecosistema .NET. Se abordan definiciones formales del enfoque, sus metas y principales ventajas, así como ejemplos concretos de cómo se aplica en estas tecnologías. También se explican los componentes básicos que forman parte de esta arquitectura. Finalmente, se presenta un caso práctico donde se muestra su aplicación en un entorno real.

2 Desarrollo

2.1 Que es clean architecture

Clean Architecture es una arquitectura de software propuesta por Robert C. Martin en 2012. En la literatura académica, se define como un estilo de arquitectura de software que agrupa varias arquitecturas orientadas a lograr la separación de responsabilidades, propone dividir el sistema y lograr una separación clara de responsabilidades entre distintas capas de una aplicación. La idea central es colocar la lógica de negocio en el núcleo del sistema y organizar alrededor de ella las capas de soporte (UI, base de datos, framework, etc.) de forma que las capas externas dependan de las internas, nunca al revés. [1].

Dicho de otro modo, la arquitectura limpia es "la unión de varias arquitecturas" que persiguen una estricta separación entre una capa de negocio y una capa de interfaces. Esta separación se rige por la llamada Regla de la Dependencia, según la cual "las dependencias del código fuente solo pueden apuntar hacia adentro; nada en un círculo interno conoce algo de un círculo externo" [1].

Un principio fundamental que sustenta Clean Architecture es la Inversión de Dependencias, una de las letras de SOLID. Este principio establece que "los detalles deben depender de abstracciones, no las abstracciones de los detalles". En la práctica, esto significa que las reglas de negocio no dependen directamente de detalles de infraestructura como la base de datos, la interfaz de usuario o frameworks; en su lugar, definen interfaces o contratos que las capas externas implementarán. Gracias a ello, el núcleo de la aplicación permanece independiente de detalles técnicos, libre de dependencias externas directas, altamente testeable y fácil de mantener en el tiempo [1].

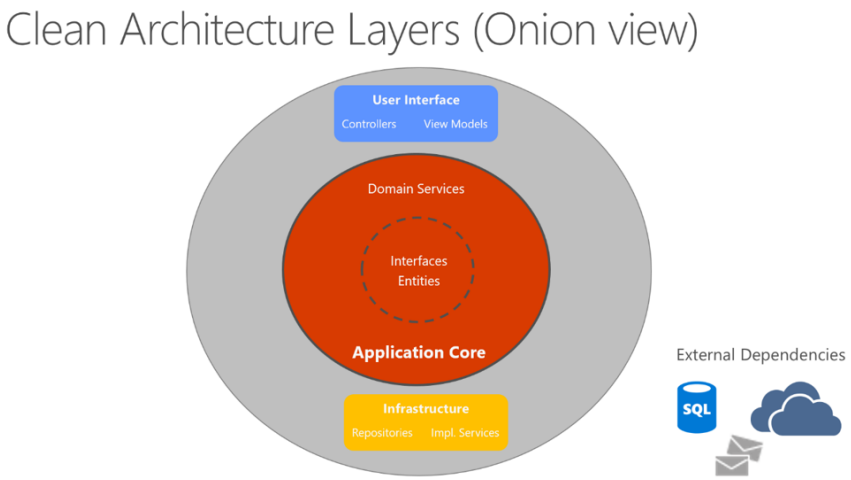

Figure 1: Representación en "vista de cebolla" de la Clean Architecture, con la lógica de negocio en el centro y las dependencias de infraestructura en las capas externas

2.2 Beneficios y Objetivos de Clean Architecture

El propósito fundamental de la Arquitectura Limpia es alcanzar una sólida separación de responsabilidades, asegurando que la lógica del dominio permanezca aislada de los elementos de infraestructura. Gracias a esto, el sistema se vuelve más comprensible, adaptable y escalable, permitiendo realizar cambios en la interfaz de usuario, la base de datos u otros módulos sin impactar el núcleo del negocio. Investigaciones académicas recientes subrayan múltiples ventajas significativas de este enfoque y son descritos a continuación:

- **Mantenibilidad y Evolución:** Al dividir responsabilidades y aislar la lógica de negocio, se facilita la evolución continua de las aplicaciones. Las modificaciones en un componente, como el sistema de base de datos o la interfaz de usuario, tienen un impacto mínimo en las demás partes del sistema, ya que cada capa gestiona su propia lógica de manera independiente. Esta organización facilita una evolución más ágil del software, reduciendo al mismo tiempo el riesgo de errores [2].

- **Separación clara y menor acoplamiento:** Una arquitectura limpia exitosa logra separar la lógica de negocio de los detalles de infraestructura [2].

- **Testabilidad mejorada:** Al mantener la lógica de negocio separada de los componentes externos, se facilita la creación de pruebas unitarias enfocadas exclusivamente en el núcleo de la aplicación, sin requerir interacción con la base de datos, la red o la interfaz de usuario. Durante las pruebas, es posible reemplazar las implementaciones reales por versiones simuladas (mocks) en las capas de infraestructura o UI, lo que simplifica y agiliza el proceso de testeo. Tal como indica la documentación oficial, al no haber dependencia directa del núcleo con la infraestructura, resulta muy sencillo desarrollar pruebas automatizadas para esta parte del sistema [2].

- **Independencia de Frameworks y Tecnologías:** La Clean Architecture tiene como propósito evitar que el diseño del sistema dependa de tecnologías específicas. Ni la interfaz de usuario, ni la base de datos, ni los frameworks utilizados condicionan la estructura del software, ya que se consideran elementos intercambiables. Esto permite, por ejemplo, reemplazar una base de datos SQL por otro sistema de almacenamiento, o cambiar la capa de presentación con un impacto mínimo en la lógica del negocio. Dicha lógica se encuentra encapsulada en un módulo independiente, sin importar si la aplicación se ejecuta en entorno web, móvil o de escritorio [2].

- **Reusabilidad y Modularidad:** Al dividir la aplicación en módulos independientes (dominio, aplicación, infraestructura, etc.), es factible reutilizar componentes en otros proyectos o reemplazarlos en caso necesario. Por ejemplo, se podría extraer la capa de dominio como una biblioteca reutilizable en múltiples aplicaciones que compartan la misma lógica de negocio [2].

- **Escalabilidad y flexibilidad:** Una arquitectura limpia bien aplicada resulta en código más flexible para cambios futuros y escalable a nuevas funcionalidades. La capa de negocio encapsulada puede evolucionar sin reescribir la interfaz o la capa de datos. Asimismo, es posible reemplazar tecnologías sin reestructurar la aplicación, siempre que las nuevas implementaciones respeten los contratos definidos en el núcleo [2].

2.3 Implementación de Clean Architecture en ASP.NET Core 9 y Blazor

En el ecosistema .NET, la adopción de Clean Architecture se apoya en la estructura modular de proyectos. La implementación típica en ASP.NET Core consiste en separar la solución en múltiples capas o proyectos, cada uno enfocado en una función específica dentro de la arquitectura.

- **Proyecto de Core:** contiene la lógica de negocio pura, por ejemplo, las entidades del dominio, interfaces de casos de uso y reglas de negocio generales.

- **Proyecto de Infrastructure:** contiene las implementaciones concretas de esas interfaces con tecnologías específicas (acceso a bases de datos con Entity Framework, llamados a servicios externos, etc.).

- **Proyecto Web o UI:** la capa más externa, que en ASP.NET Core puede ser una aplicación Web API, MVC o un frontend Blazor, encargada de la interacción con el usuario o sistemas externos.

Blazor encaja en esta arquitectura actuando como capa de presentación. Estudios recientes muestran que incluso aplicaciones Blazor Server pueden estructurarse con Clean Architecture, separando el código de UI del Core de la aplicación. Por ejemplo, en un proyecto de 2021 con Blazor, se definió un módulo “Cliente” para la interfaz, el cual referenciaba únicamente a las interfaces y expuestas por el Core, manteniendo aislada. Blazor se comunica con el servidor mediante SignalR o HTTP, pero desde el punto de vista arquitectónico simplemente constituye un front-end acoplado a la capa de Core a través de contratos bien definidos [3].

La clave para implementar esto está en aprovechar la capacidad que tiene ASP.NET Core para conectar automáticamente las diferentes partes de una aplicación mediante un mecanismo especial conocido como ”inyección de dependencias” [4].

Básicamente, en lugar de unir directamente piezas específicas, se crean contratos o acuerdos llamados ”interfaces” que definen lo que deben hacer ciertas partes de la aplicación, por ejemplo, gestionar usuarios. Luego, en un solo lugar, se indica exactamente qué clase real cumplirá con ese contrato.[4].

Cuando una parte de la aplicación, como una página web o un componente visual, necesita realizar esa tarea, solo pide la interfaz o contrato en lugar de la clase específica. ASP.NET Core automáticamente se encarga de darle la clase real adecuada [4].

De esta forma, la parte central o ”núcleo” de la aplicación nunca depende directamente de detalles específicos como bases de datos o tecnologías concretas. Esto permite crear aplicaciones limpias, flexibles y fáciles de mantener. [4].

2.4 Componentes y capas de la Clean Architecture

Una arquitectura limpia tradicional generalmente se estructura en cuatro capas concéntricas, donde cada capa reúne componentes con responsabilidades específicas dentro del sistema. A continuación, se

presentan las capas y componentes principales de la Arquitectura Limpia, así como las funciones que cumplen:

- **Entidades (Reglas de negocio de la empresa):** Las entidades representan objetos fundamentales del negocio y contienen las reglas más esenciales y abstractas del dominio. Su código está en la capa más interna, por lo que es estable e independiente de factores externos. Las entidades pueden incluir modelos del dominio, tales como clases que definen conceptos clave (por ejemplo, Usuario, Producto o Factura), así como servicios puros de dominio que agrupan lógica específica del negocio que no está ligada directamente a una entidad concreta [5]. Un aspecto esencial es que las entidades nunca dependen de capas exteriores: no interactúan directamente con bases de datos, no generan formatos como JSON y tampoco dependen de frameworks específicos. Esto garantiza que la lógica del negocio permanezca aislada, estable y fácil de probar [5].

- **Casos de Uso (Reglas de negocio de la aplicación):** Los casos de uso, también conocidos como interactores o servicios de aplicación en Clean Architecture, contienen la lógica específica que utiliza las entidades del dominio para cumplir con objetivos concretos de los usuarios. Básicamente, un caso de uso organiza cómo interactúan el mundo exterior y las entidades del negocio. Por ejemplo, en un caso llamado "Registrar Pedido", se coordinarían acciones como verificar el stock utilizando las entidades de productos, calcular el precio total, y guardar el pedido usando un repositorio [5]. En palabras simples, los casos de uso administran el flujo de información hacia las entidades y desde ellas, guiándolas para ejecutar sus reglas de negocio y alcanzar el resultado deseado. Normalmente se definen como interfaces o clases en la capa de aplicación o dominio [5].

- **Adaptadores de interfaz (Interface Adapters):** Los adaptadores de interfaz conectan la lógica del sistema con el exterior, actuando como traductores entre ambos. Adaptan la información para que los casos de uso y entidades puedan comunicarse con cosas como la interfaz de usuario o la base de datos. Aquí se encuentran los controladores y ViewModels (que reciben datos del usuario y llaman a los casos de uso) y los repositorios o mapeadores (que traducen las órdenes de los casos de uso en acciones concretas, como guardar en la base de datos). Además, convierten datos entre formatos internos y externos (como DTOs o respuestas), y cada parte (presentación y datos) está separada, sin depender una de la otra [5].

- **Frameworks y controladores de dispositivos (External interfaces):** La capa de frameworks y controladores de dispositivos es la más externa y contiene todos los elementos técnicos y herramientas externas con las que interactúa la aplicación, como bases de datos, frameworks web o de UI, APIs externas y dispositivos. Aquí va todo lo que puede cambiar sin afectar la lógica del sistema. Por ejemplo, en un proyecto con ASP.NET Core y Blazor, esta capa incluiría Blazor, ASP.NET, SQL Server, etc. También abarca código de arranque y configuración, como la clase Startup, los controladores MVC, los DbContext de Entity Framework o las páginas Razor. Esta capa se comunica con el sistema a través de adaptadores, pero las capas internas no dependen directamente de ella. Gracias a esto, se pueden reemplazar tecnologías [5].

Los distintos componentes de la Clean Architecture colaboran siguiendo un principio fundamental: las dependencias de código siempre deben dirigirse desde las capas externas hacia el núcleo, y no al contrario. Esta regla asegura que tanto las entidades como los casos de uso se mantengan independientes de los detalles técnicos. Toda comunicación con frameworks, bases de datos o dispositivos externos se realiza a través de adaptadores que implementan interfaces definidas en el núcleo. Gracias a esta estructura, se logra una clara separación de responsabilidades: las entidades definen el "qué", los casos de uso definen el "cómo", los adaptadores actúan como puente entre las capas, y los frameworks simplemente proporcionan soporte técnico.

2.5 Caso de Estudio

El repositorio de código fuente neozhu/CleanArchitectureWithBlazorServer [6], ubicado en la plataforma GitHub, se presenta como un ejemplo práctico y tangible de la aplicación de los principios de la Clean Architecture en el contexto del desarrollo de una aplicación web moderna. Esta aplicación ha sido

construida utilizando la tecnología Blazor Server, que permite la creación de interfaces de usuario interactivas con C# y .NET, y la plataforma .NET 9.0, la última iteración del robusto framework de desarrollo de Microsoft. El proyecto en sí se centra en la implementación de un panel de control de información o dashboard, una herramienta comúnmente utilizada para visualizar datos y métricas clave de una aplicación o sistema. Al abordar este tipo de aplicación, el repositorio sirve como una ilustración elocuente de cómo el código puede ser meticulosamente organizado en capas claramente delimitadas, cada una con una responsabilidad específica, adhiriéndose a los preceptos fundamentales de la Clean Architecture en un entorno de desarrollo web contemporáneo que aprovecha las capacidades avanzadas de las tecnologías .NET. La selección de este caso de estudio no es fortuita; se basa en el explícito compromiso del proyecto con la adopción de la Clean Architecture como patrón de diseño subyacente y su utilización de las tecnologías .NET más actuales, que son de particular relevancia para la presente investigación [6].

2.5.1 Análisis de Estructura de Capas

La estructura del proyecto revela las siguientes capas, alineadas con la Clean Architecture Un análisis detallado de la arquitectura interna del proyecto realizado en el contexto de los principios arquitectónicos seminales delineados por Robert C. Martin en su obra sobre la Clean Architecture [7], revela la presencia de una estructura de capas bien definida. Cada una de estas capas cumple un rol específico y esencial dentro del paradigma de la Clean Architecture, contribuyendo a la separación de preocupaciones y al cumplimiento de la regla de dependencia:

- **Dominio:** En el corazón de la aplicación reside la capa de Dominio. Este proyecto específico alberga las entidades fundamentales que sirven como modelos para los conceptos de negocio esenciales de la aplicación. Ejemplos típicos incluyen clases como User (que representa a un usuario del sistema), Order (que modela un pedido realizado) y Product (que describe un producto ofrecido). Estas clases se implementan utilizando C# puro, lo que significa que no tienen ninguna dependencia directa de los frameworks Blazor o ASP.NET Core. Esta independencia es crucial, ya que asegura que la lógica de negocio central no esté atada a detalles de implementación externos. Adicionalmente, esta capa puede contener la lógica de negocio inherente a estas entidades, definiendo su comportamiento y las reglas que rigen sus interacciones, así como las especificaciones de dominio que establecen los criterios para la validez de los datos y las operaciones.

- **Aplicación:** La capa de Aplicación se erige como el orquestador de la lógica de negocio específica que impulsa la funcionalidad de la aplicación. Este proyecto sirve como contenedor para los Casos de Uso, que a menudo se denominan servicios de aplicación o interactores. Estos componentes encapsulan la lógica necesaria para llevar a cabo tareas específicas dentro de la aplicación. Actúan como coordinadores, gestionando la interacción entre las entidades del dominio para lograr los objetivos del negocio. Un aspecto fundamental de esta capa es su utilización de las interfaces de los repositorios, cuya definición también reside en este proyecto, para abstraer la forma en que se accede a los datos persistentes. Además, la capa de Aplicación incluye los Objetos de Transferencia de Datos (DTOs), que son estructuras diseñadas para el transporte eficiente de información entre las distintas capas de la aplicación, minimizando la exposición de las entidades del dominio. Finalmente, se encuentran los mecanismos de mapeo (mappers), responsables de la conversión bidireccional entre las entidades del dominio y los DTOs, facilitando la adaptación de los datos a las necesidades de las diferentes capas. Es crucial destacar que, si bien la capa de Aplicación establece una dependencia con la capa de Dominio (ya que opera sobre las entidades y utiliza su lógica), se mantiene deliberadamente independiente de cualquier dependencia directa de los frameworks Blazor o ASP.NET Core, preservando así la portabilidad y la testabilidad de la lógica de negocio.

- **Infraestructura:** La capa de Infraestructura asume la responsabilidad de proporcionar las implementaciones concretas de las interfaces que han sido previamente definidas en la capa de Aplicación. Esto abarca la implementación de los repositorios, que son los componentes encargados de la persistencia de los datos. En el contexto de una aplicación .NET, estos repositorios a menudo utilizan tecnologías de acceso a datos específicas del ecosistema, como Entity Framework Core [8], para interactuar con los sistemas de almacenamiento de datos subyacentes. Además

de los repositorios, esta capa puede contener la lógica necesaria para la integración con servicios externos a la aplicación, como sistemas de envío de correo electrónico, APIs de terceros o cualquier otro servicio que la aplicación necesite consumir. También se encarga de la configuración particular de la base de datos, incluyendo la definición del contexto de Entity Framework Core y la configuración de las relaciones entre las entidades en el esquema de la base de datos. Es importante señalar que, a diferencia de las capas de Dominio y Aplicación, la capa de Infraestructura introduce dependencias hacia tecnologías específicas del entorno .NET, incluyendo potencialmente componentes de ASP.NET Core para tareas de configuración, como la cadena de conexión a la base de datos o la configuración de servicios específicos de la infraestructura.

- **Presentación:** La capa de Presentación se sitúa en el extremo más externo de la arquitectura y está directamente vinculada a la interfaz de usuario que se presenta al usuario final de la aplicación. En el contexto de este estudio de caso, esta capa se compone principalmente de los siguientes elementos:
  - **Blazor Server:** Los componentes de Blazor, implementados mediante archivos con extensión .razor, residen en esta capa y son los responsables primarios de la construcción de la interfaz de usuario interactiva [9]. Estos componentes definen la estructura visual de la aplicación y su comportamiento dinámico. El código C# asociado a estos componentes, ya sea directamente incrustado en el archivo .razor o a través de View Models vinculados, gestiona la lógica de presentación, es decir, cómo se muestran los datos al usuario y cómo se responde a sus interacciones. Los View Models actúan como intermediarios, preparando los datos provenientes de la capa de Aplicación en un formato adecuado para su visualización en la interfaz de usuario.
  - **ASP.NET Core 9.0:** Actúa como el entorno de host para la aplicación Blazor Server. Su rol es multifacético y abarca la gestión de las peticiones HTTP entrantes de los clientes, la configuración general de la aplicación (incluyendo la configuración de servicios y dependencias), la provisión del mecanismo de inyección de dependencias que es fundamental para la implementación de la Clean Architecture, y el establecimiento y mantenimiento de la conexión en tiempo real mediante el protocolo SignalR, que es la base de la comunicación bidireccional entre el servidor y el navegador en el modelo Blazor Server [8]. Dentro de este proyecto, el archivo Startup.cs (en versiones anteriores de .NET) o Program.cs (en versiones más recientes como .NET 6+) es el encargado de configurar los servicios necesarios para el funcionamiento de la aplicación y la cadena de middleware que procesa cada petición HTTP. En aquellos escenarios donde la aplicación expone funcionalidades a otros clientes a través de una interfaz de programación de aplicaciones (API) además de la interfaz de usuario Blazor, los Controladores de API, construidos con ASP.NET Core, también residirían en esta capa de Presentación.

2.5.2 Uso y Ubicación de Blazor y ASP.NET Core 9.0

Blazor ubicación predominante dentro de la Clean Architecture es la capa de Presentación [9]. Aquí, ejerce como la tecnología principal para la construcción de la interfaz de usuario interactiva que se presenta al usuario final. Los componentes de Blazor son los encargados de renderizar la interfaz, gestionar el estado de la UI y capturar las interacciones del usuario. Estos componentes consumen y presentan los datos proporcionados por los View Models, los cuales, a su vez, interactúan con los Casos de Uso definidos en la capa de Aplicación para obtener la información necesaria del dominio y la lógica de negocio.

ASP.NET Core 9.0: Desempeña un papel fundamental en la capa de Presentación al actuar como el host para la aplicación Blazor Server. Proporciona la infraestructura subyacente esencial para la gestión de las peticiones HTTP de los usuarios, la administración del ciclo de vida de la aplicación (inicio, apagado, etc.) y la implementación del patrón de inyección de dependencias [8]. Este último es de vital importancia para la adherencia a los principios de la Clean Architecture, ya que facilita la inversión de dependencias y permite que las capas externas (como la Presentación) dependan de abstracciones (interfaces) definidas en las capas internas (como la Aplicación) sin necesidad de conocer los detalles concretos de su implementación [7]. Adicionalmente, ASP.NET Core puede introducir dependencias en la capa de Infraestructura, particularmente a través de tecnologías como Entity Framework Core, que

requiere configuración dentro del entorno de ASP.NET Core para establecer la conexión y la interacción con los sistemas de almacenamiento de datos [8].

2.5.3 Flujo de Comunicación y Dependencias

El flujo de comunicación se adhiere de manera rigurosa al principio fundamental de la Clean Architecture que establece una dependencia unidireccional hacia las capas internas [7]:

1. El usuario inicia una interacción con la interfaz de usuario construida mediante la tecnología Blazor Server, la cual reside en la capa de Presentación [9]. Esta interacción puede ser un clic en un botón, la entrada de datos en un formulario, o cualquier otra acción que el usuario realice a través de la interfaz.

2. Los componentes de Blazor, o los View Models asociados que actúan como intermediarios, invocan los Casos de Uso que han sido definidos en la capa de Aplicación. Esta invocación se realiza a través de las interfaces que han sido inyectadas como dependencias en la capa de Presentación, gracias al soporte robusto de inyección de dependencias proporcionado por ASP.NET Core [8]. La capa de Presentación no tiene conocimiento de la implementación concreta de los Casos de Uso, solo de sus contratos definidos por las interfaces.

3. Los Casos de Uso, ubicados en la capa de Aplicación, encapsulan la lógica de negocio específica de la aplicación. Orquestan la interacción con las entidades que conforman el Dominio, aplicando las reglas de negocio y coordinando las operaciones necesarias para cumplir con la solicitud del usuario.

4. Cuando los Casos de Uso necesitan acceder a los datos persistentes o realizar operaciones relacionadas con la infraestructura (como enviar un correo electrónico), utilizan las interfaces de los Repositorios, cuya definición se encuentra también en la capa de Aplicación. De nuevo, la capa de Aplicación solo conoce los contratos definidos por estas interfaces, no la implementación subyacente.

5. La capa de Infraestructura es la responsable de implementar concretamente estas interfaces de los Repositorios. Utiliza tecnologías específicas del entorno .NET, como Entity Framework Core [8], para establecer la comunicación con los sistemas de almacenamiento de datos subyacentes y llevar a cabo las operaciones de lectura y escritura de datos. También implementa cualquier otro servicio de infraestructura requerido por la aplicación.

6. Finalmente, los datos recuperados o modificados fluyen de vuelta a través de las capas, siguiendo la dirección opuesta: desde la base de datos a través de los Repositorios implementados en la Infraestructura, hacia los Casos de Uso en la capa de Aplicación, luego a los View Models en la capa de Presentación, y culminando con su representación en la interfaz de usuario de Blazor para el usuario final.

El mecanismo de inyección de dependencias, proporcionado de manera robusta por ASP.NET Core, se erige como un pilar fundamental para el mantenimiento de una clara separación de las preocupaciones y la estricta observancia de la regla de dependencia. Permite que las capas más externas, como la Presentación y la Infraestructura, establezcan dependencias con las abstracciones (interfaces) definidas en la capa de Aplicación sin incurrir en un acoplamiento directo con sus implementaciones concretas, lo que facilita la testabilidad, la mantenibilidad y la flexibilidad de la aplicación a largo plazo [7].

2.6 Conclusiones

- En conclusión, la presente revisión sistemática destaca la Clean Architecture como un enfoque arquitectónico sólido y cada vez más adoptado para la construcción de aplicaciones web dentro del ecosistema .NET, particularmente en combinación con ASP.NET Core 9 y Blazor. Su énfasis en la separación de la lógica de negocio de las preocupaciones de infraestructura y presentación se revela como una estrategia efectiva para mitigar los desafíos inherentes al desarrollo de software a largo plazo, promoviendo la mantenibilidad, la escalabilidad y la adaptabilidad a los requisitos cambiantes.

- Para concluir, la investigación evidencia la compatibilidad y la sinergia existente entre la Clean Architecture y las tecnologías ASP.NET Core 9 y Blazor. Blazor se posiciona eficazmente como una capa de presentación que puede interactuar con el núcleo de la aplicación definido por la Clean Architecture a través de interfaces bien definidas, mientras que ASP.NET Core proporciona el entorno de ejecución y las herramientas esenciales, como la inyección de dependencias, que facilitan la implementación de los principios de esta arquitectura en el contexto del desarrollo web .NET moderno.

- Finalmente, el análisis del repositorio del caso de uso sirve como un ejemplo ilustrativo y valioso para los desarrolladores que buscan implementar la Clean Architecture en sus proyectos .NET. La organización clara en capas, el flujo de dependencias unidireccional y la separación de responsabilidades observados en este caso práctico demuestran la aplicabilidad de los conceptos teóricos de la Clean Architecture en un escenario de desarrollo web real con Blazor Server. Este estudio de caso subraya la importancia de una planificación cuidadosa y una comprensión profunda de los principios de la Clean Architecture para construir aplicaciones .NET bien estructuradas y preparadas para el futuro.

References

[1] J. F. Arias-Orezano, B. D. Reyna-Barreto, and G. Mamani-Apaza, "Repercusión de arquitectura limpia y la norma iso/iec 25010 en la mantenibilidad de aplicativos android," *TecnoLógicas*, vol. 24, no. 52, p. e2104, 2021.

[2] Microsoft Learn, "Arquitecturas comunes de aplicaciones web," 2024. Accedido el 16 de mayo de 2025.

[3] J. R. Martín, "Desarrollo de aplicación web utilizando asp.net para la gestión de reglas y sanciones en f1," 2023.

[4] J. C. Ruiz Pavón, "Nutriweb: Aplicación web para elaboración de dietas y estudio profesional," June 2021.

[5] M. L. Godoy, A. d. C. Lopez, and S. I. Mariño, "Implementación de arquitectura limpia en una aplicación móvil. registros de ingresos y egresos de personas," in *Memorias de las 53 Jornadas Argentinas de Informática e Investigación Operativa (JAIIO) - Concurso de Trabajos Estudiantiles (EST)*, (Corrientes, Argentina), pp. 115–127, Universidad Nacional del Nordeste, 2024.

[6] Neozhu, "Cleanarchitecturewithblazorserver." https://github.com/neozhu/CleanArchitectureWithBlazorServer, 2024.

[7] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. USA: Prentice Hall Press, 1st ed., 2017.

[8] Microsoft, ".net documentation," 2024. Accedido el 16 de mayo de 2025.

[9] P. Sykora, *Blazor in Action*. Manning Publications, 2020.