# ADR - Microservicios - Modelo arquitectónico del software

# ADR -DEV-BACKEND-MS

| | |
|---|---|
| Estado | VIGENTE |
| Partes interesadas | AREA GOBIERNO TECNOLOGICO SERVICIOS TRANSVERSALES |
| TAGS | |
| Tomada el | 10 jul 2024 |
| Responsable | LUIS MARTINEZ FONTIVEROS CARLOS GONZALEZ SANTIAGO |
| Documentación Asociada | Clean Architecture - Robert Martin<br>Arquitectura de Referencia SAS<br>Arquitecturas Limpias<br>Clean Code. |
| Solución Afectada | NUEVOS DESARROLLOS, PAES |

| Responsable | Aprobado por | Consultados | Informados |
|---|---|---|---|
| LUIS MARTINEZ FONTIVEROS CARLOS GONZALEZ SANTIAGO | LUIS MARTINEZ FONTIVEROS | LUIS MARTINEZ FONTIVEROS CARLOS GONZALEZ SANTIAGO | SERVICIOS TRANSVERSALES AREA GOBIERNO TECNOLOGICO<br>DESARROLLO SOFTWARE RRHH (AYESA)<br>DESARROLLO SOFTWARE ASISTENCIAL (NTTDATA)<br>DESARROLLO SOFTWARE ECONOMICO-FINANCIERO (NTTDATA) |

| | |
|---|---|
| Asunto a Decidir | El objetivo de este ADR es establecer los patrones de diseño y arquitectónicos para construir la estructura de un microservicio<br><br>Criterios:<br><br>• Que sea una solución estandarizada o al menos un estándar de facto dentro de la comunidad de desarrollo e ingeniería de software<br>• Favorece el desacoplamiento<br>• Sostenible<br>• Reutilizable<br>• Alineado con la arquitectura de referencia de contenedores.<br>• Permita el diseño basado cercano al negocio(Domain Centric ) |
| Identificación de la decisión | Se opta por una Clean Arquitecture con un adaptación a las necesidades del SAS<br><br>Para su implementación es **obligatorio seguir los principios de modularización expresados** el proyecto base que se ofrece desde el área de gobernanza para empezar a construir los microservicios. |
| Contexto | Para facilitar las implementaciones dentro de esta transformación tecnológica, se busca definir un patrón común para desarrollar la capa de acceso a datos, con el objetivo de que sea reconocible por cualquier proveedor, y en cualquier aplicación.<br>Esto permitirá al SAS, afrontar con mayores garantías situaciones como el mantenimiento de una funcionalidad, sus evolutivos, el cambio de los desarrolladores entre proyectos, la incorporación de nuevos desarrolladores o cambios de proveedor. |

## Alternativas Consideradas

### Clean Architecture

**Clean Architecture** es una arquitectura de software que promueve un diseño centrado en la separación de responsabilidades, donde las reglas de negocio (dominio) son independientes de los detalles técnicos como la interfaz de usuario, bases de datos o frameworks externos. Este enfoque asegura que las decisiones sobre los detalles tecnológicos no afecten la lógica de negocio y viceversa. Clean Architecture organiza el sistema en capas concéntricas, con la regla fundamental de que las dependencias solo pueden apuntar hacia el centro, manteniendo el dominio y los casos de uso en el núcleo, rodeados por interfaces, adaptadores y frameworks externos.

**Componentes Principales:**

- **Entidades:** Contienen las reglas de negocio de más alto nivel y son independientes de cualquier otro elemento del sistema.
- **Casos de uso:** Definen las aplicaciones y servicios específicos del sistema, utilizando las entidades para realizar las operaciones requeridas.

- **Adaptadores:** Convierten datos de un formato a otro, permitiendo la comunicación entre las capas internas y externas.
- **Frameworks y herramientas:** Incluyen elementos externos como bases de datos, interfaces de usuario y otros frameworks tecnológicos.

**Pros:**

- **Desacoplamiento Fuerte:** Las dependencias están dirigidas hacia el núcleo, lo que facilita el reemplazo de componentes sin afectar otras partes del sistema.
- **Flexibilidad y Sostenibilidad:** La clara separación de responsabilidades facilita el mantenimiento y la evolución del software a largo plazo.
- **Reutilización:** Los componentes centrales (entidades y casos de uso) son altamente reutilizables en diferentes contextos.
- **Testabilidad:** La arquitectura permite realizar pruebas unitarias fácilmente, ya que los componentes de negocio están desacoplados de los detalles técnicos.
- **Alineación con el Negocio:** El diseño se centra en las reglas de negocio y casos de uso, asegurando que el software refleje fielmente las necesidades del negocio.

**Contras:**

- **Complejidad inicial:** La implementación de Clean Architecture puede requerir una curva de aprendizaje significativa y una inversión inicial en estructurar el proyecto adecuadamente.
- **Sobrecarga:** Para proyectos pequeños o simples, la estructura de Clean Architecture puede parecer excesiva y agregar una sobrecarga innecesaria.
- **Abstracción excesiva:** La creación de múltiples capas y abstracciones puede hacer que el código sea más difícil de seguir y entender para los desarrolladores nuevos en el proyecto.

| Criterio | OK/KO | Notas |
|---|---|---|
| Estándar | OK | Aunque no es un estándar formal, Clean Architecture ha ganado popularidad y es ampliamente aceptada como una práctica recomendada en la comunidad de desarrollo de software, especialmente en el contexto de aplicaciones empresariales y sistemas complejos. |
| Reutilizable | OK | Al desacoplar las reglas de negocio de los detalles técnicos, los componentes pueden ser reutilizados en diferentes contextos y proyectos. |
| Sostenible | OK | La separación clara de responsabilidades y el enfoque en mantener las dependencias unidireccionales facilita el mantenimiento y la evolución del software a largo plazo. |
| Contenedores | OK | La separación de preocupaciones y la independencia de las dependencias externas facilitan la implementación en entornos de contenedores, permitiendo despliegues más flexibles y escalables. |
| Domain Centric | OK | El enfoque en las entidades de dominio y casos de uso asegura que el diseño esté alineado con las necesidades y reglas del negocio, promoviendo un desarrollo orientado al dominio (Domain Centric). |
| Desacoplado | OK | Promueve un fuerte desacoplamiento entre las capas, facilitando la sustitución o modificación de componentes sin afectar otras partes del sistema. |

## N-layer

**N-Layer Architecture** es una arquitectura de software que organiza una aplicación en capas separadas, donde cada capa tiene una responsabilidad específica. Las capas más comunes incluyen la capa de presentación, la capa de lógica de negocio y la capa de acceso a datos. Esta estructura permite una separación clara de preocupaciones, facilitando el desarrollo, mantenimiento y escalabilidad del sistema. Cada capa se comunica únicamente con la capa adyacente, asegurando una interacción organizada y minimizando el acoplamiento.

**Componentes Principales:**

- **Capa de presentación:** Maneja la interfaz de usuario y la interacción con el cliente. Recibe las entradas del usuario y las pasa a la capa de lógica de negocio.
- **Capa de lógica de negocio:** Contiene la lógica central de la aplicación y las reglas de negocio. Procesa los datos recibidos de la capa de presentación y envía los resultados a la capa de acceso a datos.
- **Capa de acceso a datos:** Gestiona la comunicación con la base de datos u otros servicios de almacenamiento, realizando operaciones CRUD (crear, leer, actualizar, eliminar).
- **Capa de servicios (opcional):** Puede añadirse para manejar la lógica específica de servicios externos, integraciones y API.

**Pros:**

- **Simplicidad y Familiaridad:** Es una arquitectura bien conocida y comprendida, lo que facilita su adopción y uso.
- **Separación de Preocupaciones:** Cada capa tiene una responsabilidad específica, lo que organiza el código de manera lógica y facilita el desarrollo y mantenimiento.
- **Modularidad:** Las capas bien definidas permiten modificar una capa sin afectar directamente a las otras, promoviendo la modularidad del sistema.
- **Reutilización:** Los componentes dentro de una capa pueden ser reutilizados en diferentes partes del sistema.

**Contras:**

- **Desacoplamiento limitado:** Aunque separa preocupaciones técnicas, el acoplamiento entre capas puede convertirse en un problema si no se maneja adecuadamente, especialmente en sistemas grandes y complejos.
- **Rígido en estructura:** La estricta separación en capas puede dificultar la implementación de ciertas funcionalidades que requieren interacción más fluida entre diferentes aspectos del sistema.
- **Escalabilidad limitada:** Puede ser menos flexible en términos de escalabilidad y evolución del sistema en comparación con enfoques más modernos como Clean Architecture.
- **Menos enfoque en el negocio:** Tiende a enfocarse más en la separación técnica que en la lógica y necesidades del negocio, lo que puede llevar a una arquitectura menos alineada con los objetivos del negocio.

| Criterio | OK/KO | Notas |
|---|---|---|
| Estándar | OK | La arquitectura en capas es una práctica bien establecida y ampliamente utilizada en la industria del software, conocida y comprendida por muchos desarrolladores |
| Reutilizable | OK | Las capas bien definidas permiten reutilizar componentes dentro de la misma capa en diferentes partes del sistema |
| Sostenible | OK | La estructura en capas facilita el mantenimiento, permitiendo cambios en una capa sin afectar directamente a las demás. |
| | | |

| | | |
|---|---|---|
| Contenedores | KO | Aunque no tan inherentemente alineada como Clean Architecture, puede adaptarse a entornos de contenedores con una configuración adecuada. |
| Domain Centric | KO | No siempre se alinea de forma tan clara con el diseño basado en el dominio (Domain Centric), ya que a menudo se centra más en la separación técnica que en la lógica de negocio. |
| Desacoplado | KO | Proporciona un nivel básico de desacoplamiento al separar la lógica de negocio, la lógica de presentación y el acceso a datos en diferentes capas. |

**Tabla Comparativa**

| Criterio | Clean Architecture | N-Layers |
|---|---|---|
| Estándar | Más moderna y emergente, reconocida pero no un estándar formal. | Más tradicional y ampliamente aceptada como una práctica común en la industria. |
| Reutilizable | Alta reutilización debido a la independencia de componentes. | Buena reutilización dentro de capas, pero menos flexible. |
| Sostenible | Muy sostenible a largo plazo gracias a su estructura y separación clara de responsabilidades. | Sostenible pero puede enfrentar problemas de mantenimiento en sistemas complejos. |
| Contenedores | Mejor alineada debido a su independencia de dependencias externas. | Puede alinearse con contenedores, pero requiere una configuración adecuada. |
| Domain Centric | Fuerte enfoque en el diseño centrado en el dominio y las necesidades del negocio. | Más centrada en la separación técnica, menos alineada con las necesidades específicas del negocio. |
| Desacoplado | Fuerte desacoplamiento y dependencia unidireccional. | Desacoplamiento básico, pero puede volverse problemático en sistemas grandes.. |

## Decisión Tomada

La arquitectura elegida es Clean Architecture.

Es la opción que encaja  mejor en los criterios:

- La que aporta una mayor sostenibilidad
- La que aporta un mayor acercamiento al dominio
- Y el que aporta un mayor desacoplamiento, especialmente

Para su implementación es **obligatorio seguir los principios de modularización expresados** el proyecto base que se ofrece desde el área de gobernanza para empezar a construir los microservicios.

- Domain
- Application
- Rest
- Message

Se han de seguir los aspectos fundamentales asociados a buenas prácticas expresados en los documentos siguientes:

- Arquitecturas Limpias.

- Clean Code.

## Consecuencias

> **Alcance de esta sección**
>
> Las consecuencias deben ceñirse a la realidad del proyecto. Las consecuencias expresadas a continuación sirven de guía en un marco de generalidad sin aplicación específica al proyecto y equipo de desarrollo.

- **Contras**
  - El conocimiento y la implementación de esta arquitectura puede conllevar una curva de aprendizaje por parte de los equipos, sobre todo aquellos que no cuenten con una persona con suficiente experiencia para poder abordarlo.
  - El entendimiento del proyecto base puede conllevar una curva de aprendizaje por parte de los equipos, sobre todo aquellos que no cuenten con una persona con suficiente experiencia para poder abordarlo.
- **Pros**
  - Arquitectura orientado a negocio y no a la implementación, lo que lo hace menos vulnerables a cambios tecnológicos (migraciones o actualizaciones) que no conlleven un cambio funcional asociado.
  - Una vez aprendido, al ser una Arquitectura y estandarizarse, aumentará la sostenibildad y el onboarding en los proyectos que tengan implementado el patrón
  - El ejemplo está implementado basándose puramente en el estándar de microprofile **(Vendor Neutral)** y no en frameworks externos , Esto garantiza su sostenibilidad en el tiempo y la compatibilidad de múltiples servidores de aplicaciones y motores de bbdd sin tener que hacer adaptaciones o implementaciones específicas.
  - Una vez aprendido:
    - Aumentará la velocidad de desarrollo, ya que los equipos no tendrán que implementarse en cada microservicio la capa de acceso a datos, solo los elementos específicos de cada negocio
    - Mejorará el onboarding ante la rotación dentro de los equipos o cambios de proveedor
    - Mejorará la sosteniblidad ya que sólo tiene dependencias con el estándar que son mucho más estables en el tiempo.

## Estado Actual

Los trabajos iniciados antes de la fecha de entrada en vigor se registrarán como deuda técnica. Cualquier trabajo iniciado posteriormente será evaluado obligatoriamente para garantizar su cumplimiento.