

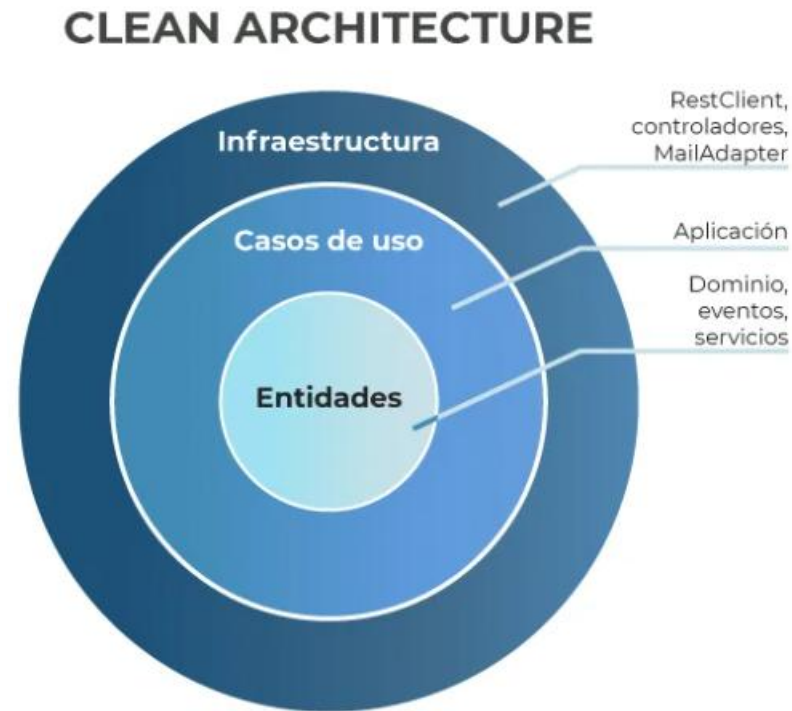
## View Models

- Reglas de negocio
- Models
- View Models



# Arquitecturas Limpias

La **arquitectura limpia** (*Clean Architecture*), propuesta por **Robert C. Martin (Uncle Bob)**, es un modelo de diseño de software que busca **mantener el código desacoplado, mantenible y adaptable** ante los cambios tecnológicos.



# Arquitecturas Limpias

## Concepto de dominio

El dominio es el corazón del problema que el software intenta resolver. Representa la realidad del negocio, sus conceptos, reglas, procesos y relaciones, independientemente de la tecnología usada para implementarlo.

Dicho más simple:

El dominio es “de qué trata” un sistema y cómo funcionan las cosas dentro de ese mundo.

# Arquitecturas Limpias

## Elementos del dominio de un software

El dominio incluye:

**Entidades (entities)** : los objetos principales del negocio (por ejemplo, Libro, Socio, Préstamo).

**Objeto de Valor (Value Objects)**: datos que tienen significado pero no identidad (por ejemplo, una FechaDeVencimiento o un Monto).

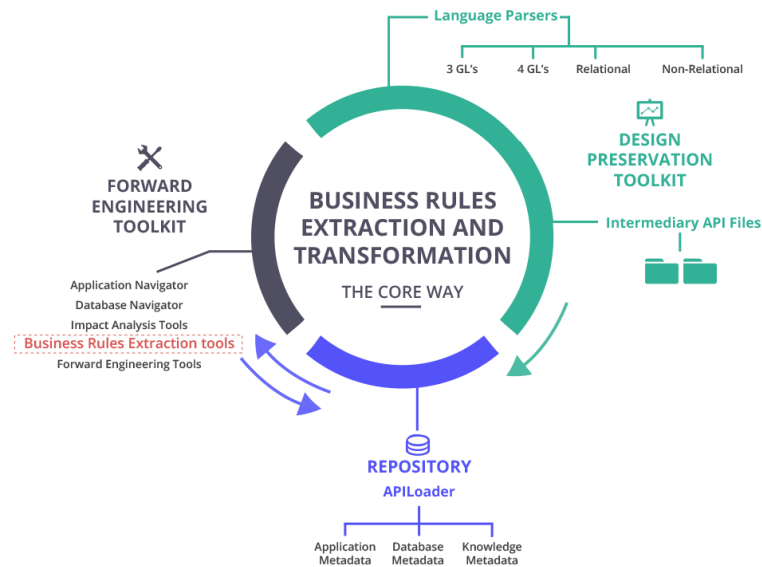
**Reglas de negocio (business logic)**: cómo deben comportarse esas entidades (por ejemplo, “un socio no puede tener más de tres préstamos activos”).

**Servicios de dominio o casos de uso**: coordinan acciones entre entidades (por ejemplo, “registrar un préstamo”).

# Lógica de negocio

El término **reglas de negocio/ lógica de negocio** (o *business logic*) se refiere al conjunto de **reglas, decisiones y procesos** que describen cómo funciona el dominio del problema que el software está resolviendo.

Estas reglas, definen o limitan algún aspecto del comportamiento del sistema y son esenciales para el correcto funcionamiento de una aplicación. Pueden abordar una variedad de aspectos, desde la validación de datos hasta la lógica de procesos.



# Lógica de negocio

**Por ejemplo,** si un sistema gestiona pedidos de un restaurante, la lógica de negocio incluye saber que *un pedido no puede cerrarse si no tiene al menos un plato confirmado*, o que *el total debe incluir los impuestos aplicables*.



# Modelos

Las reglas de negocio se utilizan para garantizar que el software opere de acuerdo con los requisitos y las políticas establecidas. Para esto se suele armar el modelo o núcleo de una aplicación

El modelo de arquitectura limpia separa la lógica de negocio de:

- Los periféricos de entrada/salida
- Sintaxis SQL
- Estructuras de datos que se importan de una librería de tercero
- El motor que renderiza las plantillas html
- El formato en que se debe enviar la respuesta al cliente ya sea JSON o XML

# Arquitecturas Limpias

## Ejercicio

Tu aplicación gestiona una **biblioteca pública**.

El sistema debe permitir registrar **libros, socios y préstamos**.



# Reglas de negocio (Modelos)

¿Cuál sería un límite razonable para la cantidad de libros que un socio puede tomar prestados simultáneamente?

**Regla:** Un socio puede tomar prestados hasta 5 libros al mismo tiempo.

¿Qué debería ocurrir si alguien intenta sacar un libro que ya está prestado a otro socio?

**Regla:** El sistema debe notificar que el libro está prestado y no permitir el préstamo hasta que sea devuelto.

¿Durante cuánto tiempo debería un socio poder conservar un libro antes de devolverlo?

**Regla:** El período de préstamo es de 14 días desde la fecha de retiro del libro.

¿Y si un socio devuelve el libro fuera de plazo? ¿Debe el sistema tomar alguna medida?

**Regla:** En caso de devolución tardía, se aplicará una multa diaria hasta que el libro sea devuelto.

¿Qué pasa si un socio con multas impagas intenta pedir otro libro?

**Regla:** No se podrá registrar un préstamo a un socio que tenga multas pendientes.

¿Hay algún tipo de libro que solo pueda consultarse dentro de la biblioteca y no llevarse a casa?

**Regla:** Ciertos libros, como los de referencia, no pueden retirarse del edificio.

# Reglas de negocio (Modelos)

**Todo eso que acabamos de decir:** un socio no puede tener más de tres libros, que no se puede prestar un libro si ya está en uso es la **lógica de negocio**.

**Todo lo demás, es decir:** cómo lo guardo, cómo lo muestro, qué base uso son **detalles técnicos** que pueden cambiar sin afectar esas reglas.

# Definiendo de aquí

**Entidades** : en el sistema son los objetos del dominio, los conceptos centrales que existen en la realidad del negocio. Tienen atributos y comportamientos propios, y persisten en el tiempo (no dependen de una operación puntual).

Ejemplo (biblioteca):

- ***Libro, Socio, Préstamo.***
- Un **Libro** tiene *título, autor, ISBN*.
- Un **Socio** tiene *nombre, número de carnet, etc.*

En código, las entidades suelen ser clases del dominio con métodos que representan comportamientos válidos (por ejemplo, `prestamo.vencer()`, `socio.agregarMulta()`).

# Definiendo de aquí

## **Reglas de negocio: *Cómo deben comportarse esas entidades***

Son las **normas o restricciones** que determinan qué está permitido y qué no dentro del dominio.

Las reglas de negocio son **la lógica que gobierna** a las entidades.

Ejemplo (biblioteca):

- Un socio **no puede tener más de 3 préstamos activos.**
- Un préstamo **vence a los 15 días.**
- Si el libro se devuelve tarde, **se genera una multa.**

# Definiendo de aquí

Los **modelos** son estructuras que usamos para **transportar o adaptar información**, generalmente hacia o desde las capas externas del sistema.

No necesariamente representan las reglas del negocio, sino **una forma de organizar los datos** para una tarea específica.

Ejemplo (biblioteca):

- Un **LibroDTO** para enviar datos del libro por una API.
- Un **LibroViewModel** que incluye formato de fecha o etiquetas para la interfaz.
- Un **LibroRecord** que representa cómo se guarda en la base de datos.

# Definiendo de aquí

Los **modelos** son estructuras que usamos para **transportar o adaptar información**, generalmente hacia o desde las capas externas del sistema.

No necesariamente representan las reglas del negocio, sino **una forma de organizar los datos** para una tarea específica.

Ejemplo (biblioteca):

- Un **LibroDTO** para enviar datos del libro por una API.
- Un **LibroViewModel** que incluye formato de fecha o etiquetas para la interfaz.
- Un **LibroRecord** que representa cómo se guarda en la base de datos.

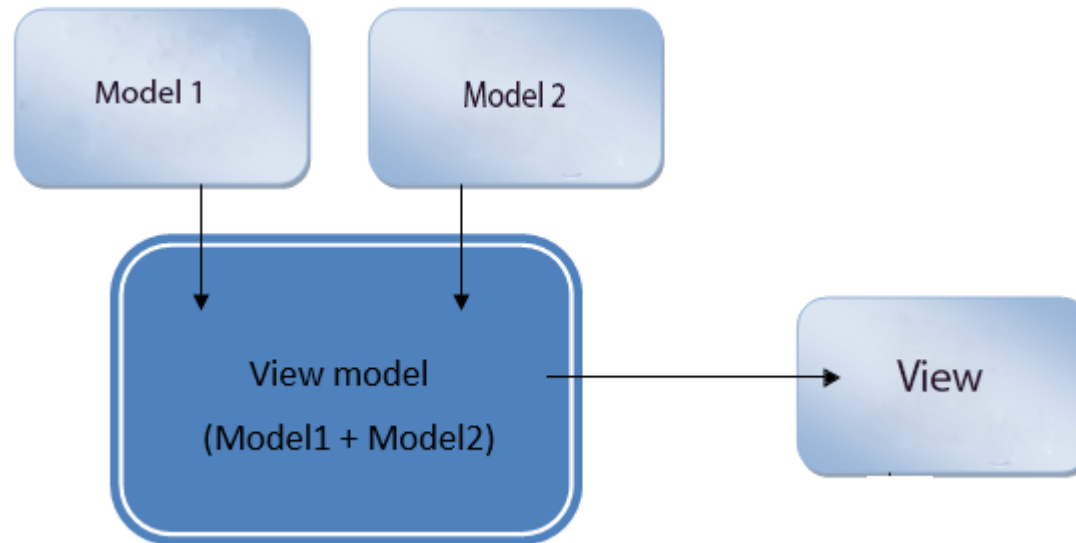
# Definiendo de aquí

El modelo puede contener los mismos datos que una entidad, pero **sin la lógica de negocio**. Es una versión *plana*, pensada para transporte o persistencia.

# Modelo de Vista (View models)

En ciertas situaciones, es posible que un solo objeto de modelo no contenga todos los datos necesarios para una vista, o por el contrario necesite más datos para una determinada Vista.

En tales situaciones, necesitamos usar ViewModel en la aplicación.



# Modelo de Vista (View models)

Llamamos ViewModel al objeto que se le pasa a una vista para facilitar la presentación de los datos. Normalmente se busca que estos objetos sean los más simples posibles y que tengan solo datos relacionados con la necesidad de la vista.

En pocas palabras, los View Models son los equivalentes de los modelos para un sistema pero orientado a una vista.

# Comparando – models vs view models

## Models

**Abstracciones que** Representa los datos reales del dominio o de la base de datos.

Suele reflejar directamente entidades de negocio o tablas, y está más cerca del “núcleo” del sistema.

## View models

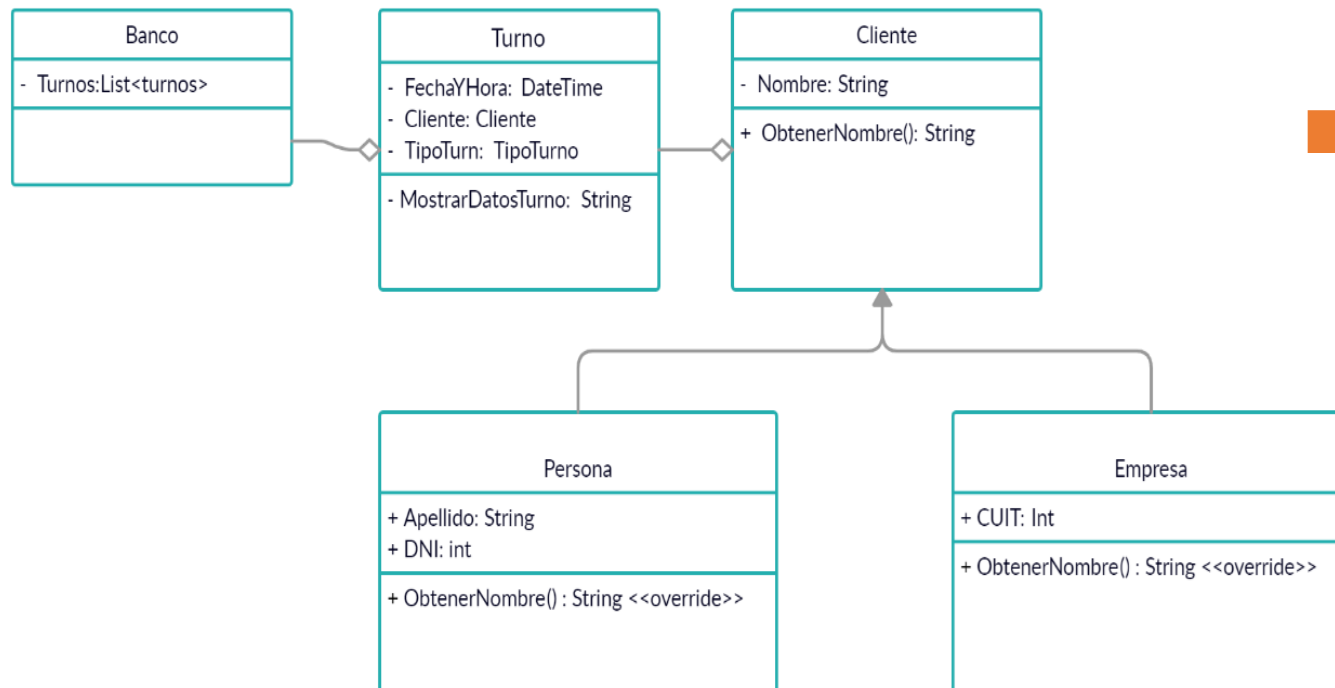
**Abstracciones que** facilitan la presentación y manipulación de datos en una vista.

Un ViewModel (Modelo de Vista) es una estructura diseñada específicamente para una interfaz o pantalla concreta.

Su propósito es mostrar o recolectar datos de la vista (UI), no representar la lógica del negocio ni la base de datos.

# Comparando – models vs view models

Turno en el backend (modelo)



Turno en la vista  
(TurnoViewModel)

Fecha y hora  
Nombre cliente

# View models

## Escenario 1: Simplificando un Modelo Complejo a un ViewModel más simple

```
public class Producto
{
    public int Id { get; set; }
    public string Nombre { get; set; }
    public string Descripcion { get; set; }
    public decimal Precio { get; set; }
    public DateTime FechaDeAlta { get; set; }
    public DateTime FechaDeAlta { get; set; }
}
```



```
public class ProductoSimpleViewModel
{
    public string Nombre { get; set; }
    public decimal Precio { get; set; }

    public ProductoSimpleViewModel(Producto producto)
    {
        Nombre = producto.Nombre;
        Precio = producto.Precio;
    }
}
```

# View models

Escenario 2: Usando Dos Modelos para Llegar a un ViewModel

```
public class Usuario
{
    public int Id { get; set; }
    public string Nombre { get; set; }
}

public class HistorialCompras
{
    public int Usuariold { get; set; }
    public decimal MontoTotalCompras { get; set; }
}
```



```
public class DetallesUsuarioViewModel
{
    public string Nombre { get; set; }
    public decimal MontoTotalCompras { get; set; }

    public DetallesUsuarioViewModel(Usuario usuario, HistorialCompras historialCompras)
    {
        Nombre = usuario.Nombre;
        MontoTotalCompras = historialCompras.MontoTotalCompras;
    }
}
```

# View models

## Escenario 3: Modelo Simple con ViewModel que Contiene Más Datos

```
public class Libro
{
    public int Id { get; set; }
    public string Titulo { get; set; }
    // Otras propiedades del libro
}
```



```
public class DetallesLibroViewModel
{
    public string Titulo { get; set; }
    public string Autor { get; set; }
    public string Categoria { get; set; }

    public DetallesLibroViewModel(Libro libro, string autor, string
categoria)
    {
        Titulo = libro.Titulo;
        Autor = autor;
        Categoria = categoria;
    }
}
```

# View models

## Puntos Importantes

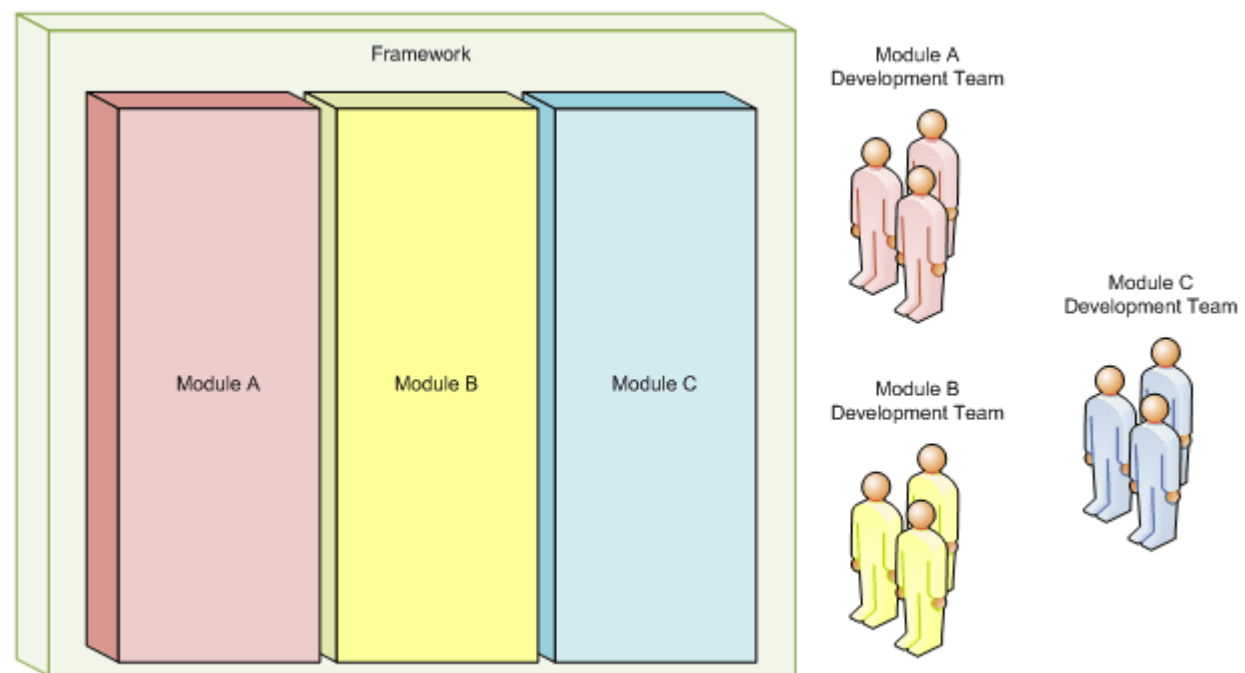
- Contiene campos que están representados en la vista (para Helpers LabelFor,EditorFor,DisplayFor)
- Puede tener reglas de validación específicas mediante anotaciones de datos o IDataErrorInfo.
- Poner solo datos/campos relevantes a la Vista que se quiere representa.
- Si solo se utilizan los campos de los view models, esto facilitará la renderización y el mantenimiento

# Bibliografía

<https://docs.microsoft.com/es-es/dotnet/api/system.componentmodel.dataannotations?view=net-5.0>

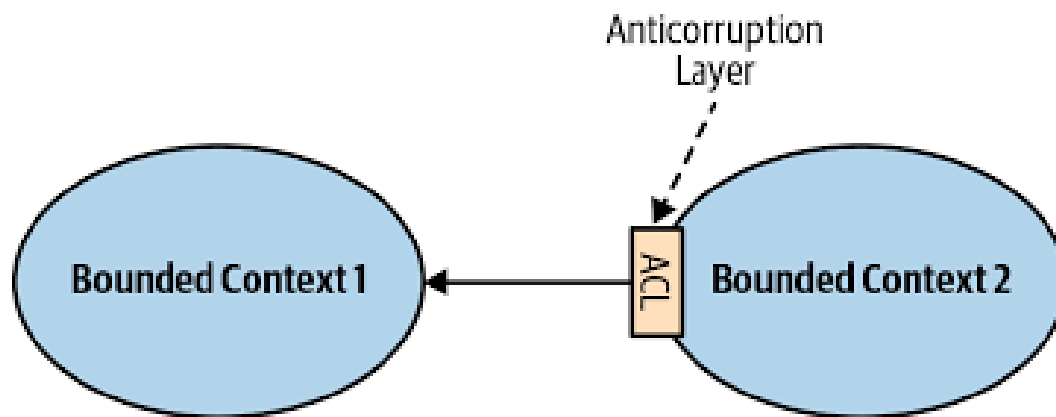
<https://wakeupandcode.com/validation-in-asp-net-core-3-1/>

<https://www.thereformedprogrammer.net/wrapping-your-business-logic-with-anti-corruption-layers-net-core/>



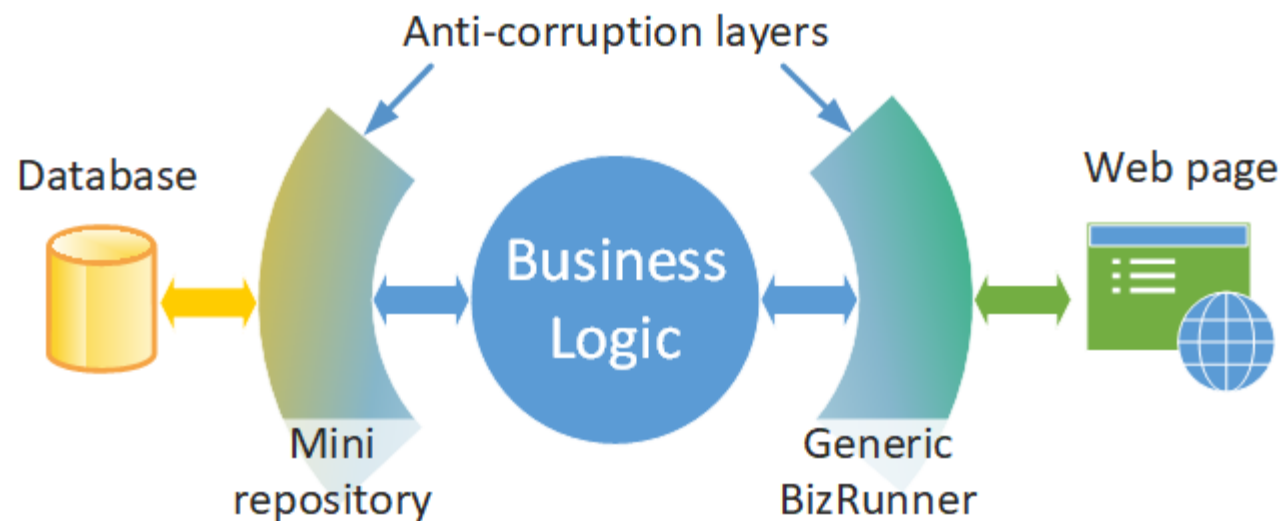
# Capa anticorrupción

El Diseño Dirigido por el Dominio (DDD) define una capa anti-corrupción como un patrón de adaptador que aísla una parte de un sistema, conocida en DDD como un "contexto delimitado", de otro contexto delimitado. Su función es garantizar que la semántica en un contexto delimitado no "corrompa" la semántica del otro contexto delimitado.



# Capa anticorrupción

Según [la documentación de Microsoft](#) “la capa anticorrupción Implementa una fachada o capa de adaptador entre diferentes subsistemas que no comparten la misma semántica.”



# Capa anticorrupción

Algunos ejemplos de uso de una capa anti-corrupción en una aplicación web son:

**Lógica de negocios:** Su lógica de negocios no debería tener que preocuparse por cómo funciona la base de datos o el front-end.

**Front-end:** Los usuarios “humanos” necesitan datos que les sean útiles, no necesariamente conocer cómo se diseñó la base de datos o las estructuras internas de un sistema.

**Web API:** Su API web debe proporcionar un servicio que se ajuste a las necesidades externas.

**Seguridad:** Las capas anti-corrupción también pueden ayudar con la seguridad, al pasar únicamente los datos exactos que la otra área necesita.