



# Implementando Diseño impulsado por el dominio

Una guía práctica para implementar el diseño impulsado por el dominio con el marco ABP

Halil Ibrahim Kalkan

# Implementando Diseño impulsado por el dominio

Una guía práctica para implementar el Dominio  
Diseño Dirigido con el Marco ABP

**Halil Ibrahim Kalkan**

Autor:	Halil Ibrahim Kalkan
Diseñador:	Melis Latino
Fecha de publicación:	Junio de 2021 (Primera edición)

The Official Guide

# Mastering ABP Framework

Building maintainable .NET solutions by following software development best practices using ABP



By  
Halil Ibrahim KALKAN

You can buy from

**amazon**

# CONTENIDO

Introducción .....	4
Meta .....	5
Código simple .....	6
¿Qué es el diseño impulsado por el dominio? .....	7
OOP Y SÓLIDO .....	7
Capas DDD y arquitectura limpia .....	8
Elementos básicos básicos .....	10
Implementación: El panorama general .....	12
Capas de una solución .NET .....	12
Dependencias de los Proyectos en la Solución .....	18
Flujo de ejecución de una aplicación basada en DDD .....	22
Principios comunes .....	23

# CONTENIDO

Implementación: Los bloques de construcción .....	28
El dominio de ejemplo .....	28
Agregados .....	29
Repositorios .....	52
Especificaciones .....	57
Servicios de dominio .....	64
Servicios de aplicación .....	68
Objetos de transferencia de datos .....	71
Ejemplos de casos de uso .....	82
Creación de entidad .....	82
Actualización/manipulación de una entidad .....	91
Lógica de dominio y lógica de aplicación .....	95
Capas de aplicación múltiples .....	96
Ejemplos .....	98
Libros de referencia .....	106

# INTRODUCCIÓN

Esta es una **guía práctica** para implementar el diseño impulsado por el dominio (DDD). Si bien los detalles de implementación se basan en la infraestructura del marco ABP, los conceptos, principios y patrones centrales son aplicables a cualquier tipo de solución, incluso si no se trata de una solución .NET.

# Meta

Los objetivos de este libro son:

- Presentar y explicar la arquitectura, los conceptos, los principios, los patrones y los bloques de construcción del DDD.
- Explicar la arquitectura en capas y la estructura de la solución que ofrece el marco ABP.
- Introducir reglas explícitas para implementar patrones DDD y mejores prácticas dando ejemplos concretos.
- Muestre lo que ABP Framework le ofrece como Infraestructura para implementar DDD de manera adecuada.
- Y, por último, brindar sugerencias basadas en las mejores prácticas de desarrollo de software y nuestras experiencias para crear una base de código mantenable.

# ¡Código simple!

Jugar al fútbol es muy sencillo, pero jugar al fútbol sencillo es lo más difícil que existe. — Johan Cruyff

Si tomamos esta famosa frase de programación, podemos decir;

Escribir código es muy sencillo, pero escribir código simple es lo más difícil que existe. — ???

En este documento presentaremos **reglas simples y fáciles de implementar**.

Una vez que su aplicación crezca, será difícil seguir estas reglas. A veces , romper las reglas le ahorrará tiempo a corto plazo. Sin embargo, el tiempo ahorrado a corto plazo traerá consigo una pérdida de tiempo mucho mayor a mediano y largo plazo. Su base de código se vuelve complicada y difícil de mantener. La mayoría de las aplicaciones comerciales se reescriben simplemente porque ya no se pueden mantener .

Si sigue las reglas y las mejores prácticas, su base de código será más simple y fácil de mantener. Su aplicación reaccionará a los cambios más rápido.

## ¿Qué es el diseño impulsado por el dominio?

El diseño impulsado por el dominio (DDD) es un enfoque para el desarrollo de software para **necesidades complejas** que conecta la implementación a un modelo **en evolución** ;

DDD es adecuado para **dominios complejos y aplicaciones a gran escala**, en lugar de aplicaciones CRUD simples. Se centra en la **lógica del dominio central** en lugar de en los detalles de la infraestructura.

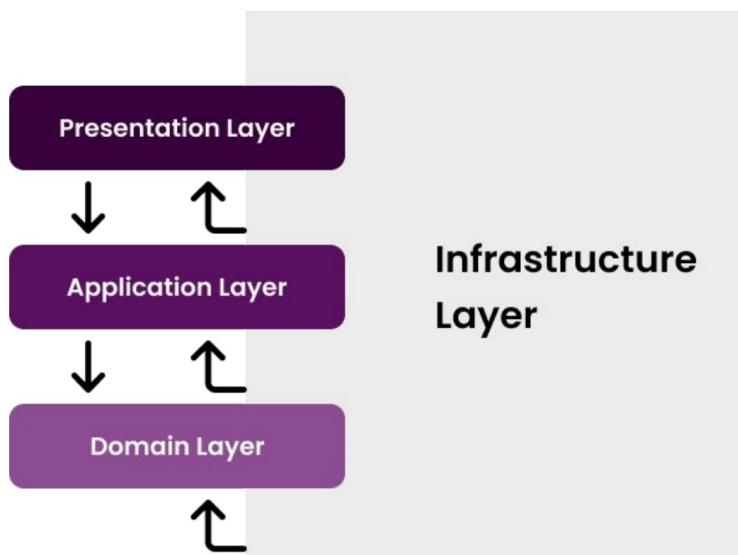
Ayuda a construir una **base de código flexible, modular y mantenible** .

## OOP y SÓLIDO

La implementación de DDD depende en gran medida de la Programación Orientada a Objetos (POO) y **SOLID** Principios. En realidad, **implementa y extiende** estos principios. Por lo tanto, una **buenas comprensión** de OOP y SOLID lo ayuda mucho a la hora de implementar realmente el DDD.

# Capas DDD y arquitectura limpia

Hay cuatro capas fundamentales de un dominio impulsado  
Solución basada en;



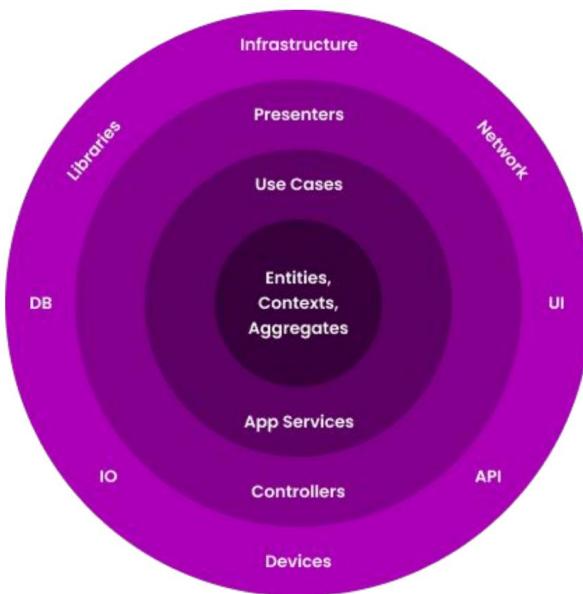
La lógica empresarial se divide en dos capas, la capa de dominio y la capa de aplicación, que contienen diferentes tipos de lógica empresarial;

- La capa de dominio implementa el núcleo, el caso de uso lógica de negocio independiente del dominio/sistema.
- La capa de aplicación implementa los casos de uso de la aplicación en función del dominio. Un caso de uso puede considerarse como una interacción del usuario en la interfaz de usuario (IU).

- La capa de presentación contiene los elementos de la interfaz de usuario (páginas, componentes) de la aplicación.
- La capa de infraestructura respalda a otras capas mediante Implementar las abstracciones e integraciones con sistemas y bibliotecas de terceros.

La misma superposición se puede mostrar como el diagrama a continuación y se conoce como Arquitectura Limpia o , a veces, como Cebolla.

Arquitectura:



En la Arquitectura Limpia, cada capa depende únicamente de la capa que se encuentra directamente dentro de ella. La capa más independiente se muestra en el círculo más interno y es la Capa de Dominio.

## Bloques de construcción básicos

La DDD se centra principalmente en las capas de dominio y aplicación e ignora la presentación y la infraestructura. Se consideran detalles y las capas empresariales no deberían depender de ellos.

Eso no significa que las capas de presentación e infraestructura no sean importantes. Son muy importantes. Los marcos de interfaz de usuario y los proveedores de bases de datos tienen sus propias reglas y prácticas recomendadas que debes conocer y aplicar. Sin embargo, estas no están dentro de los temas de DDD.

Esta sección presenta los componentes básicos esenciales de las capas de dominio y aplicación.

## Bloques de construcción de la capa de dominio

- **Entidad:** Una entidad es un objeto con sus propias propiedades (estado, datos) y métodos que implementan la lógica empresarial que se ejecuta en estas propiedades. Una entidad se representa mediante su identificador único (Id). Dos objetos de entidad con diferentes Id se consideran entidades diferentes.
- **Objeto de valor:** Un objeto de valor es otro tipo de objeto de dominio que se identifica por sus propiedades en lugar de por un identificador único. Esto significa que dos objetos de valor con las mismas propiedades se consideran el mismo objeto. Los objetos de valor generalmente se implementan como inmutables y, en su mayoría, son mucho más simples que las entidades.

- **Agregado y raíz de agregado:** un **agregado** es un conjunto de objetos (entidades y objetos de valor) unidos por un objeto raíz agregado. La **raíz agregada** es un tipo específico de entidad con algunas responsabilidades adicionales.
- **Repositorio (interfaz):** Un **repositorio** Es una interfaz de tipo colección que utilizan las capas de dominio y aplicación para acceder al sistema de persistencia de datos (la base de datos). Oculta la complejidad del DBMS del código comercial. La capa de dominio contiene las **interfaces** de los repositorios.
- **Servicio de dominio:** un **servicio de dominio** es un servicio sin estado que implementa las reglas de negocio básicas del dominio. Resulta útil implementar la lógica del dominio que depende de varios tipos de agregados (entidades) o de algunos servicios externos.
- **Especificación:** Una **especificación** Se utiliza para definir filtros con nombre, reutilizables y combinables para entidades y otros objetos comerciales.
- **Evento de dominio:** un **evento de dominio** es una forma de informar a otros servicios de manera flexible cuando ocurre un evento específico del dominio.

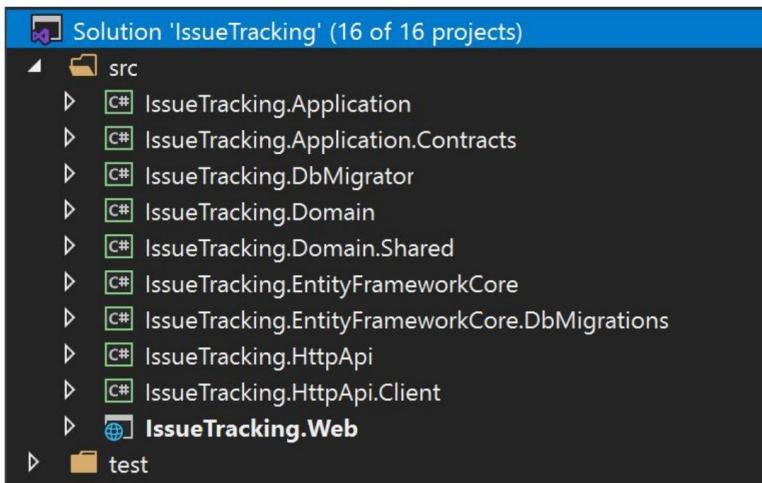
## Bloques de construcción de la capa de aplicación

- Servicio de aplicación: un servicio de aplicación es un servicio sin estado que implementa casos de uso de la aplicación. Un servicio de aplicación normalmente obtiene y devuelve DTO. Lo utiliza la capa de presentación. Utiliza y coordina los objetos de dominio para implementar los casos de uso. Un caso de uso normalmente se considera una unidad de trabajo.
- Objeto de transferencia de datos (DTO): un DTO es un objeto simple sin ninguna lógica de negocios que se utilice para transferir estado (datos) entre las capas de aplicación y presentación.
- Unidad de trabajo (UOW): Una unidad de trabajo es una obra atómica. Esto debería realizarse como una unidad de transacción. Todas las operaciones dentro de una UOW deberían confirmarse en caso de éxito o revertirse en caso de falla.

## Implementación: el panorama general

### Capas de una solución .NET

La siguiente imagen muestra una solución de Visual Studio creada utilizando la plantilla de inicio de la aplicación de ABP:



El nombre de la solución es **IssueTracking** y consta de múltiples proyectos. La solución se estructura en capas considerando **los principios DDD** así como **prácticas de desarrollo e implementación**. El sub Las secciones siguientes explican los proyectos en la solución;

La estructura de su solución puede ser ligeramente diferente si elige una interfaz de usuario o un proveedor de base de datos diferente. Sin embargo, las capas de dominio y aplicación serán las mismas y este es el punto esencial para la perspectiva DDD. Consulte la [plantilla de inicio de la aplicación Documento](#) si desea saber más sobre la estructura de la solución.

## La capa de dominio

La capa de dominio se divide en dos proyectos;

- **IssueTracking.Domain** es la capa de dominio esencial que contiene todos los bloques de construcción (entidades, objetos de valor, servicios de dominio, especificaciones, interfaces de repositorio, etc.) introducidos anteriormente.
- **IssueTracking.Domain.Shared** es un proyecto delgado que Contiene algunos tipos que pertenecen a la capa de dominio, pero que se comparten con todas las demás capas. Por ejemplo, puede contener algunas constantes y enumeraciones relacionadas con los objetos de dominio, pero que otras capas deben reutilizar.

## La capa de aplicación

La capa de aplicación también se divide en dos proyectos;

- **IssueTracking.Application.Contracts** contiene el Interfaces de servicio de aplicaciones y los DTO que utilizan estas interfaces. Este proyecto puede ser compartido por las aplicaciones cliente (incluida la interfaz de usuario).
- **IssueTracking.Application** es la capa de aplicación esencial que implementa las interfaces definidas en el proyecto Contratos.

## La capa de presentación

- `IssueTracking.Web` es un MVC ASP.NET Core / Razor Pages Aplicación para este ejemplo. Este es el único ejecutable. aplicación que sirve a la aplicación y a las API.

ABP Framework también admite diferentes tipos de marcos de interfaz de usuario, incluido [Angular](#). y [Blazor](#). En estos casos, `IssueTracking.Web` no existe en la solución. En su lugar, habrá una aplicación `IssueTracking.HttpApi.Host` en la solución para brindar servicio a las API HTTP como un punto final independiente que las aplicaciones de la interfaz de usuario pueden utilizar a través de llamadas a la API HTTP.

## La capa de servicio remoto

- El proyecto `IssueTracking.HttpApi` contiene las API HTTP definidas por la solución. Normalmente contiene **controladores MVC** y modelos relacionados, si están disponibles. Por lo tanto, escribe sus API HTTP en este proyecto.

La mayoría de las veces, los controladores de API son simplemente envoltorios de los servicios de aplicación para exponerlos a los clientes remotos. Dado que [el sistema de controlador de API automático](#) de ABP [Framework](#)

Configura y expone automáticamente sus servicios de aplicación como controladores de API. Normalmente, no crea controladores en este proyecto. Sin embargo, la solución de inicio lo incluye para los casos en los que necesita crear controladores de API manualmente.

- El proyecto [IssueTracking.HttpApi.Client](#) es útil cuando Tengo una aplicación C# que necesita consumir su HTTP. API. Una vez que la aplicación cliente hace referencia a este proyecto, puede [inyectar](#) [directamente](#) y utilizar los servicios de la aplicación. Esto es posible con la ayuda del [C# dinámico](#) del marco ABP [Sistema de Proxies API de Cliente.](#)

Hay una aplicación de consola en la carpeta [de prueba](#) de la solución, denominada [IssueTracking.HttpApi.Client.ConsoleTestApp](#). Simplemente utilice el proyecto [IssueTracking.HttpApi.Client](#) para consumir las API expuestas por la aplicación. Es solo una aplicación de demostración y puede eliminarla de forma segura. Incluso puede eliminar el proyecto [IssueTracking.HttpApi.Client](#) si cree que no lo [necesita.](#)

## La capa de infraestructura

En una implementación de DDD, puede tener un solo proyecto de infraestructura para implementar todas las abstracciones e integraciones, o puede tener diferentes proyectos para cada dependencia.

Sugerimos un enfoque equilibrado: crear proyectos separados para las principales dependencias de infraestructura (como Entity Framework Core) y un proyecto de infraestructura común para el resto de infraestructura.

La solución startup de ABP cuenta con dos proyectos para la Entidad Integración del núcleo del marco;

- **IssueTracking.EntityFrameworkCore** es esencial

Paquete de integración para EF Core. **El DbContext de su aplicación**, las asignaciones de bases de datos, las implementaciones de los repositorios y otros elementos relacionados con EF Core se encuentran aquí.

- **IssueTracking.EntityFrameworkCore.DbMigrations** es un proyecto especial para administrar las migraciones de bases de datos de Code First. Hay un **DbContext independiente** en este proyecto para realizar un seguimiento de las migraciones. Normalmente, no se toca mucho este proyecto, excepto cuando se necesita crear una nueva migración de base de datos o agregar un **módulo de aplicación**, que tiene algunas tablas de base de datos y naturalmente requiere crear una nueva migración de base de datos.

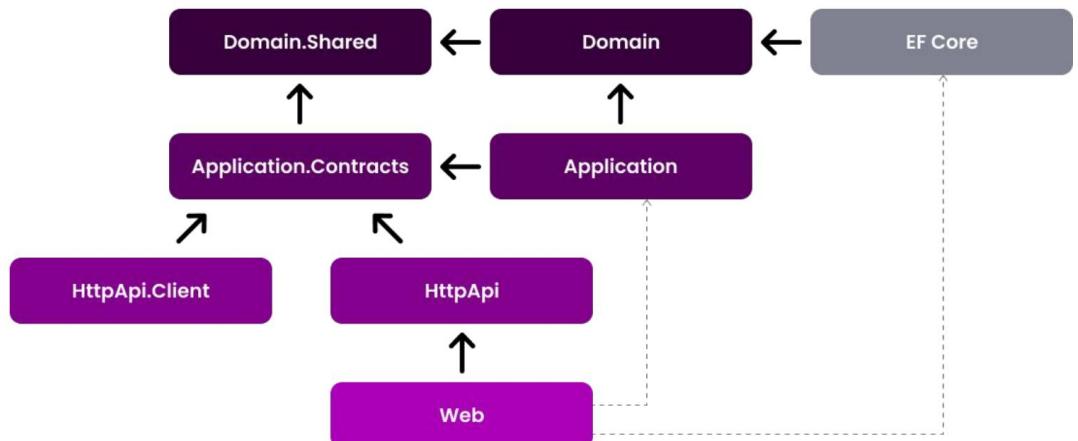
Quizás te preguntes por qué hay dos proyectos para EF Core. La cuestión está relacionada principalmente con la modularidad. Cada módulo tiene su propio `DbContext` independiente y su aplicación también tiene un `DbContext`. El proyecto DbMigrations contiene una unión de los módulos para realizar un seguimiento y aplicar una única ruta de migración. Si bien la mayoría de las veces no necesita saberlo, puede consultar el [EF Core migraciones](#) Documento para obtener más información.

## Otros proyectos

Hay un proyecto más, `IssueTracking.DbMigrator`, que es una aplicación de consola sencilla que migra el esquema de la base de datos y semillas los datos iniciales cuando lo ejecutas. Es una herramienta útil Aplicación de utilidad que puedes utilizar tanto en desarrollo como en entorno de producción.

## Dependencias de los Proyectos en la Solución

El diagrama siguiente muestra las dependencias esenciales (proyecto referencias) entre los proyectos en la solución (`IssueTracking`). (no se muestra que la parte sea simple)



Los proyectos ya se han explicado anteriormente. Ahora podemos explicar las razones de las dependencias:

- **Domain.Shared** es el proyecto que todos los demás proyectos dependen directa o indirectamente de. Por lo tanto, todos los tipos de este proyecto están disponibles para todos los proyectos.
- El **dominio** solo depende de **Domain.Shared** porque ya es una parte (compartida) del dominio. Por ejemplo, una enumeración **IssueType** en **Domain.Shared** puede ser utilizada por una entidad **Issue** en el proyecto **Domain**.
- **Application.Contracts** depende de **Domain.Shared**. De esta manera, puede reutilizar estos tipos en los DTO. Por ejemplo, la misma enumeración **IssueType** en **Domain.Shared** puede ser utilizada por un **CreateIssueDto** como propiedad.

- La aplicación depende de `Application.Contracts`, ya que implementa las interfaces de los servicios de aplicación y utiliza los DTO dentro de ella. También depende del dominio, ya que los servicios de aplicación se implementan utilizando los objetos de dominio definidos dentro de él.
- `EntityFrameworkCore` depende del dominio ya que asigna los objetos del dominio (entidades y tipos de valores) a las tablas de la base de datos (ya que es un ORM) e implementa las interfaces del repositorio definidas en el dominio.
- `HttpApi` depende de `Application.Contracts` ya que el Los controladores dentro de él inyectan y utilizan las interfaces del servicio de aplicación como se explicó anteriormente.
- `HttpApi.Client` depende de `Application.Contracts` ya que puede consumir los servicios de aplicación como se explicó anteriormente.
- La Web depende de `HttpApi` ya que sirve las API HTTP definidas dentro de ella. También, de esta manera, depende indirectamente del proyecto `Application.Contracts` para consumir los Servicios de Aplicación en las Páginas/Componentes.

## Dependencias discontinuas

Cuando investigue la solución, verá dos dependencias más mostradas con líneas discontinuas en la figura anterior.

El proyecto `web` depende de los proyectos `Application` y `EntityFrameworkCore`, que en teoría no deberían ser así, pero en realidad lo son.

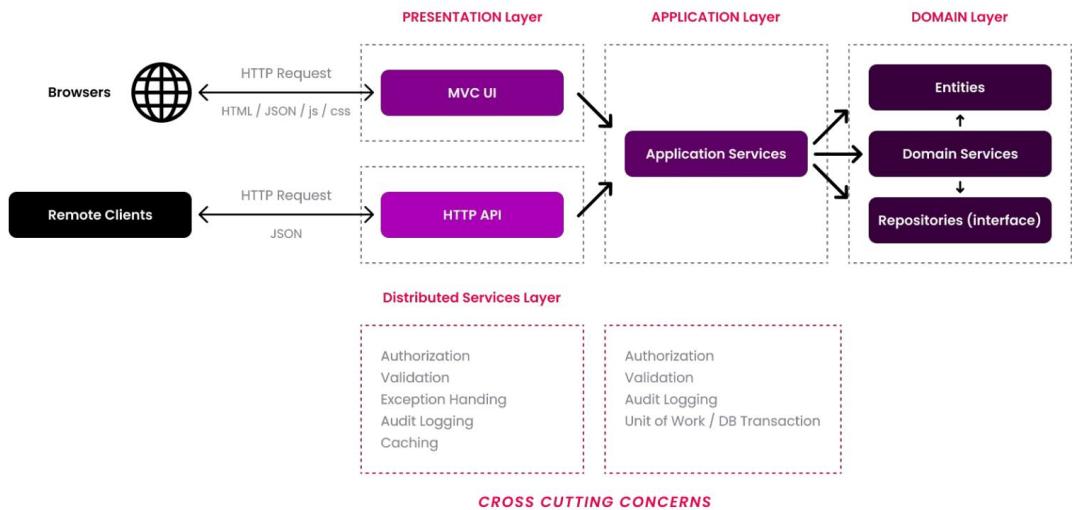
Esto se debe a que la `Web` es el proyecto final que ejecuta y aloja la aplicación y la aplicación necesita las implementaciones de los Servicios de Aplicación y los Repositorios mientras se ejecuta.

Esta decisión de diseño potencialmente le permite utilizar entidades y objetos EF Core en la capa de presentación, lo que **debería evitarse estrictamente**. Sin embargo, consideramos que los diseños alternativos son demasiado complicados. Aquí, dos de las alternativas si desea eliminar esta dependencia:

- Convierte el proyecto `web` en una biblioteca de clases de Razor y crea un nuevo proyecto, como `WebHost`, que depende de los proyectos `web`, `de aplicación` y `EntityFrameworkCore` y aloja la aplicación. No escribes ningún código de interfaz de usuario aquí, sino que lo usas solo para el alojamiento.
- Eliminar la aplicación y `EntityFrameworkCore` Dependencias del proyecto `web` y carga sus ensamblados al inicializar la aplicación. Puede utilizar `los módulos de complemento de ABP sistema para tal fin.`

## Flujo de ejecución de una aplicación basada en DDD

La siguiente figura muestra un flujo de solicitud típico para una aplicación web que se ha desarrollado en base a patrones DDD.



- La solicitud normalmente comienza con una interacción del usuario en la interfaz de usuario (un caso de uso) que genera una solicitud HTTP al servidor.
- Un controlador MVC o un controlador de página Razor en el La capa de presentación (o capa de servicios distribuidos) maneja la solicitud y puede realizar algunas tareas transversales en esta etapa (Autorización, Validación, Manejo de excepciones, etc.). Un controlador/página inyecta la interfaz de servicio de aplicación relacionada y llama a sus métodos enviando y recibiendo DTO.

- El servicio de aplicación utiliza los objetos de dominio (Entidades, interfaces de repositorio, servicios de dominio, etc.) para implementar el caso de uso. La capa de aplicación implementa algunas cuestiones transversales (autorización, validación, etc.). Un método de servicio de aplicación debe ser una **unidad de trabajo**. Eso significa que debería ser atómico.

La mayoría de las cuestiones transversales se **implementan de forma automática** y convencional mediante el marco ABP y, por lo general, no es necesario escribir código para ellas.

## Principios comunes

Antes de entrar en detalles, veamos algunos principios generales de DDD;

### Proveedor de base de datos / Independencia de ORM

Las capas de dominio y de aplicación deben ser independientes del ORM y del proveedor de bases de datos. Solo deben depender de las interfaces del repositorio, y estas no deben utilizar ningún objeto específico de ORM.

A continuación se exponen las principales razones de este principio:

1. Para crear la infraestructura de su dominio/aplicación independiente ya que la infraestructura puede cambiar en el futuro o puede ser necesario admitir un segundo tipo de base de datos más adelante.
2. Hacer que su dominio/aplicación se centre en el código comercial ocultando los detalles de la infraestructura detrás de los repositorios.
3. Para hacer tus pruebas automatizadas más fáciles ya que en este caso puedes simular los repositorios.

Respecto a este principio, ninguno de los proyectos de la solución tiene referencia al proyecto EntityFrameworkCore , excepto la aplicación de inicio.

## Discusión sobre la independencia de la base de datos Principio

En particular, la razón 1 afecta profundamente el diseño de objetos de su dominio (especialmente, las relaciones entre entidades) y el código de la aplicación.

Supongamos que está utilizando Entity Framework Core con una base de datos relacional. Si desea que su aplicación sea compatible con MongoDB Más adelante, no podrás utilizar algunas funciones muy útiles de EF Core.

Ejemplos:

- No se puede asumir [el seguimiento de cambios Dado](#) que el proveedor MongoDB no puede hacerlo, siempre es necesario actualizar explícitamente las entidades modificadas.
- No puedes usar [las Propiedades de navegación \(o Colecciones\)](#) a Otros agregados en sus entidades, ya que esto no es posible para una base de datos de documentos. Consulte la "Regla: Referencia a otros Sección "Agregados sólo por Id"" para obtener más información.

Si cree que estas características son **importantes** para usted y **nunca se alejará de** EF Core, creemos que vale la pena **ampliar este principio**. Seguimos sugiriendo utilizar el patrón de repositorio para ocultar los detalles de la infraestructura. Pero puede asumir que está utilizando EF Core mientras diseña sus relaciones de entidad y escribe el código de su aplicación. Incluso puede hacer referencia al paquete NuGet de EF Core desde su capa de aplicación para poder utilizar directamente los métodos de extensión LINQ asincrónicos, como [\*\*ToListAsync\(\)\*\*](#) (consulte la **sección IQueryble & Async Operations en [Repositorios](#)**). documento para más información).

---

## Tecnología de presentación agnóstica

La tecnología de presentación (marco de interfaz de usuario) es una de las partes más modificadas de una aplicación del mundo real. Es muy importante diseñar las **capas de dominio y aplicación** para que sean completamente **ajenas** a la tecnología/marco de presentación. Este principio es relativamente fácil de implementar y la plantilla de inicio de ABP lo hace aún más fácil.

En algunos casos, es posible que necesite tener **lógica duplicada** en las capas de aplicación y presentación. Por ejemplo, es posible que necesite duplicar las **comprobaciones de validación y autorización** en ambas capas. Las comprobaciones en la capa de IU son principalmente para **la experiencia del usuario**, mientras que las comprobaciones en las capas de aplicación y dominio son para **la seguridad y la integridad de los datos**. Eso es perfectamente normal y necesario.

## Centrarse en los cambios de estado, no en los informes

DDD se centra en cómo **cambian los objetos del dominio y sus interacciones**; cómo crear una entidad y cambiar sus propiedades preservando la integridad/valididad de los datos e implementando las **reglas de negocio**.

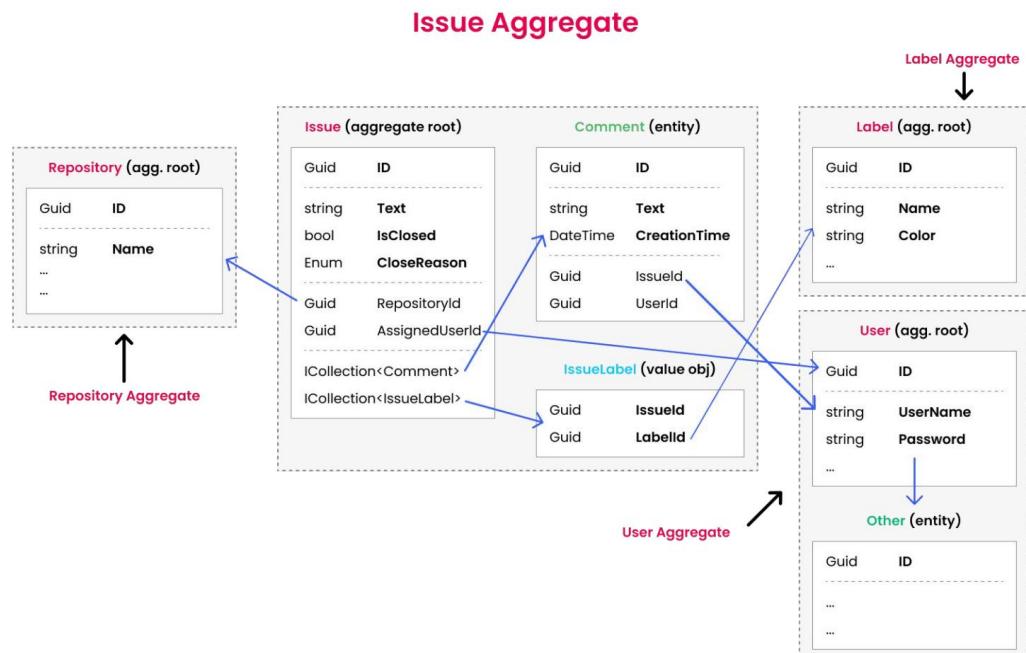
DDD ignora los informes y las consultas masivas. Eso no significa que no sean importantes. Si su aplicación no tiene paneles de control ni informes sofisticados, ¿quién los usaría? Sin embargo, los informes son otro tema. Por lo general, desea utilizar toda la potencia de SQL Server o incluso utilizar una fuente de datos independiente (como ElasticSearch) para fines de informes. Escribirá consultas optimizadas, creará índices e incluso procedimientos almacenados (!). Tiene la libertad de hacer todas estas cosas siempre que no las infecte en su lógica empresarial.

# Implementación: los elementos básicos

Esta es la parte esencial de esta guía. Presentaremos y explicaremos algunas **reglas explícitas** con ejemplos. Puede seguir estas reglas y aplicarlas en su solución mientras implementa el diseño impulsado por el dominio.

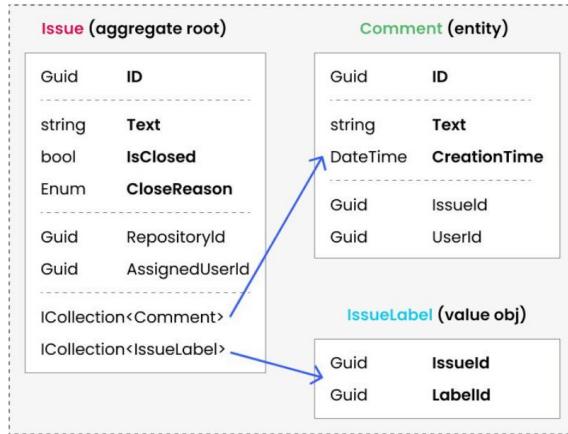
## El dominio de ejemplo

Los ejemplos utilizarán algunos conceptos que utiliza GitHub, como **Issue**, **Repository**, **Label** y **User**, con los que ya estás familiarizado. La siguiente figura muestra algunos de los agregados, raíces de agregados, entidades, objetos de valor y las relaciones entre ellos:



El agregado de problemas consta de una raíz de agregado de problemas que contiene colecciones de comentarios y etiquetas de problemas. Otros agregados se muestran como simples, ya que nos centraremos en el agregado de problemas:

## Issue Aggregate



## Agregados

Como se dijo antes, un Agregado es un grupo de objetos (entidades y objetos de valor) unidos entre sí por un objeto raíz agregado.

En esta sección se presentarán los principios y reglas relacionados con los Agregados.

Nos referimos al término Entidad tanto para entidades raíz agregadas como para entidades de subcolección, a menos que escribamos explícitamente entidad raíz agregada o de subcolección.

# Principios de agregados y raíces de agregados

## Reglas de negocio

Las entidades son responsables de implementar las reglas de negocio relacionadas con sus propias propiedades. Las entidades raíz agregadas también son responsables de sus entidades de subcolección.

Un agregado debe mantener su **integridad y validez** mediante la implementación de reglas y restricciones de dominio. Esto significa que, a diferencia de los DTO, las entidades tienen **métodos para implementar cierta lógica empresarial**. En realidad, deberíamos intentar implementar reglas empresariales en las entidades siempre que sea posible.

## Unidad individual

Un agregado se **recupera y se guarda como una sola unidad**, con todas las subcolecciones y propiedades. Por ejemplo, si desea agregar un **comentario** a un **problema**, debe:

- Obtener el **problema** de la base de datos incluyendo todas las subcolecciones (**comentarios** y **etiquetas de problema**).
- Utilice métodos en la **clase Issue** para agregar un nuevo comentario, como **Issue.AddComment(...);**
- Guardar el **problema** (con todas las subcolecciones) en la base de datos como una única operación de base de datos (actualización).

Esto puede parecer extraño para los desarrolladores que solían trabajar con EF Core y bases de datos relacionales . Obtener el problema con todos los detalles parece innecesario e inefficiente. ¿Por qué no ejecutamos simplemente un comando SQL Insert en la base de datos sin consultar ningún dato?

La respuesta es que debemos implementar las reglas de negocio y preservar la coherencia e integridad de los datos en el código. Si tenemos una regla de negocio como "Los usuarios no pueden comentar en el

Problemas bloqueados", ¿cómo podemos comprobar el estado de bloqueo de un problema sin recuperarlo de la base de datos? Por lo tanto, podemos ejecutar las reglas de negocio solo si los objetos relacionados están disponibles en el código de la aplicación.

Por otro lado, los desarrolladores de MongoDB encontrarán esta regla muy natural. En MongoDB, un objeto agregado (con subcolecciones) se guarda en una única colección en la base de datos (mientras que se distribuye en varias tablas en una base de datos relacional). Por lo tanto, cuando se obtiene un agregado, todas las subcolecciones ya se recuperan como parte de la consulta, sin ninguna configuración adicional.

ABP Framework ayuda a implementar este principio en sus aplicaciones.

## Ejemplo: Agregar un comentario a un problema

```
public class IssueAppService : ApplicationService, IIssueAppService
{
    private readonly IRepository<Issue, Guid> _issueRepository;

    public IssueAppService(IRepository<Issue, Guid> issueRepository)
    {
        _issueRepository = issueRepository;
    }

    [Authorize]
    public async Task CreateCommentAsync(CreateCommentDto input)
    {
        var issue = await _issueRepository.GetAsync(input.IssueId);
        issue.AddComment(CurrentUser.GetId(), input.Text);
        await _issueRepository.UpdateAsync(issue);
    }
}
```

El método `_issueRepository.GetAsync` recupera el `Issue` con todos los detalles (subcolecciones) como una sola unidad de forma predeterminada. Si bien esto funciona de manera inmediata para MongoDB, debe configurar los detalles agregados para EF Core. Pero, una vez que lo configure, los repositorios lo manejarán automáticamente. El método `_issueRepository.GetAsync` obtiene un parámetro opcional, `includeDetails`, que puede pasar `como falso` para deshabilitar este comportamiento cuando lo necesite.

Consulte la sección Carga de entidades relacionadas del [núcleo EF documento](#) para la configuración y escenarios alternativos.

`Issue.AddComment` obtiene un ID de usuario y un texto de comentario.

Implementa las reglas de negocio necesarias y agrega las comentar en la colección Comentarios del Número.

Finalmente, usamos `_issueRepository.UpdateAsync` para guardar los cambios. A la base de datos.

EF Core tiene una función de seguimiento de cambios , por lo que en realidad no es necesario llamar a `_issueRepository.UpdateAsync`. Se guardará automáticamente gracias al sistema de unidad de trabajo de ABP, que llama automáticamente a `DbContext.SaveChangesAsync()` al final del método. Sin embargo, en el caso de MongoDB, es necesario actualizar explícitamente la entidad modificada.

Entonces, si desea escribir su código independientemente del proveedor de base de datos, siempre debe llamar al método `UpdateAsync` para las entidades modificadas.

## Límite de transacción

Un agregado generalmente se considera como un límite de transacción.

Si un caso de uso trabaja con un solo agregado, lo lee y lo guarda como una sola unidad, todos los cambios realizados en los objetos agregados se guardan juntos como una operación atómica y no es necesario realizar una transacción de base de datos explícita.

Sin embargo, en la vida real, es posible que necesite cambiar más de una instancia **agregada** en un solo caso de uso y necesite usar transacciones de base de datos para garantizar **la actualización atómica y la coherencia de los datos**. Por eso, ABP Framework utiliza una transacción de base de datos explícita para un caso de uso (un límite de método de servicio de aplicación). Consulte la [Unidad de trabajo](#) documentación para obtener más información.

## Serializabilidad

Un agregado (con la entidad raíz y las subcolecciones) debe ser serializable y transferible en la red como una sola unidad. Por ejemplo, MongoDB serializa el agregado en un documento JSON mientras lo guarda en la base de datos y lo deserializa desde JSON mientras lee desde la base de datos.

Este requisito no es necesario cuando se utilizan bases de datos relacionales y ORM. Sin embargo, es una práctica importante del diseño orientado al dominio.

Las siguientes reglas ya traerán la serializabilidad.

## Reglas y mejores prácticas para agregados y raíces agregadas

Las siguientes reglas garantizan la implementación de los principios introducidos anteriormente.

### Referencia a otros agregados únicamente por ID

La primera regla dice que un agregado debe hacer referencia a otros agregados únicamente por su ID. Esto significa que no se pueden agregar propiedades de navegación a otros agregados.

- Esta regla permite implementar la serialización.  
principio.
- También evita que diferentes agregados se manipulen entre sí y filtren la lógica empresarial de un agregado a otro.

En el siguiente ejemplo verás dos raíces agregadas, **GitRepository** y **Issue** ;

```
public class GitRepository : AggregateRoot<Guid>
{
    public string Name { get; set; }

    public int StarCount { get; set; }

    public Collection<Issue> Issues { get; set; } X
}

public class Issue : AggregateRoot<Guid>
{
    public string Text { get; set; }

    public GitRepository Repository { get; set; } X

    public Guid RepositoryId { get; set; } ✓
}
```

- **GitRepository** no debería tener una colección de **problemas** ya que son agregados diferentes.
- El **problema** no debe tener una propiedad de navegación para el **GitRepository** relacionado ya que es un agregado diferente.
- El **problema** puede tener **RepositoryId** (como **Guid**).

Entonces, cuando tienes un **problema** y necesitas tener un **GitRepository** relacionado con este problema, necesitas consultarla explícitamente desde la base de datos por **RepositoryId**.

## Para bases de datos relacionales y básicas de EF

Naturalmente, en MongoDB no es adecuado tener este tipo de propiedades o colecciones de navegación. Si lo hace, encontrará una copia del objeto agregado de destino en la colección de la base de datos del agregado de origen, ya que se serializa en JSON al guardar.

Sin embargo, los desarrolladores de EF Core y bases de datos relacionales pueden considerar innecesaria esta regla restrictiva, ya que EF Core puede manejarla en la lectura y escritura de bases de datos. Consideramos que esta es una regla importante que ayuda a **reducir la complejidad** del dominio y previene problemas potenciales, y recomendamos enfáticamente implementarla. Sin embargo, si cree que es práctico ignorar esta regla, consulte la **sección Discusión sobre el principio de independencia de la base de datos** que se encuentra más arriba.

## Mantenga los agregados pequeños

Una buena práctica es mantener el agregado simple y pequeño. Esto se debe a que un agregado se cargará y guardará como una sola unidad y la lectura y escritura de un objeto grande presenta problemas de rendimiento. Vea el siguiente ejemplo:

```
public class Role : AggregateRoot<Guid>
{
    public string Name { get; set; }

    public Collection<UserRole> Users { get; set; } X
}

public class User : AggregateRoot<Guid>
{
    public string Name { get; set; }

    public Collection<UserRole> Roles { get; set; } ✓
}
```

```
public class UserRole : ValueObject
{
    public Guid UserId { get; set; }

    public Guid RoleId { get; set; }
}
```

El agregado de roles tiene una colección de objetos de valor **UserRole** para realizar un seguimiento de los usuarios asignados para este rol. Tenga en cuenta que **UserRole** no es otro agregado y no es un problema para la regla Referenciar otros agregados solo por Id. Sin embargo, es un problema en la práctica. Un rol puede asignarse a miles (incluso millones) de usuarios en un escenario de la vida real y es un problema de rendimiento significativo cargar miles de elementos cada vez que se consulta un **rol** desde la base de datos (recuerde: los agregados se cargan por sus subcolecciones como una sola unidad).

Por otro lado, el **Usuario** puede tener dicha colección de Roles , ya que un usuario no tiene muchos roles en la práctica y puede ser útil tener una lista de roles mientras trabaja con un Agregado de Usuarios.

Si piensas detenidamente, hay un problema más cuando Role y User tienen la lista de relaciones cuando se utiliza una base de datos no relacional, como

MongoDB. En este caso, la misma información se duplica en diferentes colecciones y será difícil mantener la coherencia de los datos (siempre que agregues un elemento a **User.Roles**, también debes agregarlo a **Role.Users** ).

Por lo tanto, determine los límites y el tamaño del agregado basándose en las siguientes consideraciones:

- Objetos utilizados juntos.
- Rendimiento de consultas (cargar/guardar) y consumo de memoria.
- Integridad, validez y consistencia de los datos.

En la práctica;

- La mayoría de las raíces agregadas no tendrán subcolecciones.

- Una subcolección no debe tener más de 100-150

elementos dentro de ella en el máximo de los casos. Si cree que una colección potencialmente puede tener más elementos, no defina la colección como parte del agregado y considere extraer otra raíz agregada para la entidad dentro de la colección.

## Claves primarias en las raíces/entidades agregadas

- Una raíz agregada normalmente tiene una única propiedad **Id** para su identificador (Primaria Key: PK). Preferimos **Guid** como PK de una entidad raíz agregada (consulte el [documento Generación de Guid](#) para saber por qué).
- Una entidad (que no es la raíz del agregado) en un agregado puede usar una clave primaria compuesta.

Por ejemplo, vea la raíz agregada y la entidad a continuación:

### Aggregate Root

Define a **single** Primary Key (**Id**)

```
public class Organization
{
    public Guid Id { get; set; }

    public string Name { get; set; }

    //...
}
```

### Entity

Can define a **composite** Primary Key

```
public class OrganizationUser
{
    public Guid OrganizationId { get; set; }

    public Guid UserId { get; set; }

    public bool IsOwner { get; set; }

    //...
}
```

- La organización tiene un identificador Guid (Id).
- OrganizationUser es una subcolección de una Organización y tiene una clave principal compuesta que consta de OrganizationId y UserId.

Esto no significa que las entidades de subcolección siempre deban tener claves principales compuestas. Pueden tener propiedades de identificación única cuando sea necesario.

Las claves primarias compuestas son en realidad un concepto de bases de datos relacionales, ya que las entidades de subcolección tienen sus propias tablas y necesitan una clave primaria. Por otro lado, por ejemplo, en MongoDB no es necesario definir una clave primaria para las entidades de subcolección, ya que se almacenan como parte de la raíz agregada.

## Constructores de las raíces/entidades agregadas

El constructor se encuentra donde comienza el ciclo de vida de una entidad. Un constructor bien diseñado tiene algunas responsabilidades:

- Obtiene las propiedades de entidad requeridas como parámetros para crear una entidad válida. Debe forzar el paso solo de los parámetros requeridos y puede obtener propiedades no requeridas como parámetros opcionales.

- Comprueba la validez de los parámetros.
- Inicializa subcolecciones.

## Ejemplo de constructor de Problema (Raíz agregada)

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using Volo.Abp;
using Volo.Abp.Domain.Entities;

namespace IssueTracking.Issues
{
    public class Issue : AggregateRoot<Guid>
    {
        public Guid RepositoryId { get; set; }
        public string Title { get; set; }
        public string Text { get; set; }
        public Guid? AssignedUserId { get; set; }
        public bool IsClosed { get; set; }
        public IssueCloseReason? CloseReason { get; set; } //enum

        public ICollection<IssueLabel> Labels { get; set; }

        public Issue(
            Guid id,
            Guid repositoryId,
            string title,
            string text = null,
            Guid? assignedUserId = null
        ) : base(id)
        {
            RepositoryId = repositoryId;
            Title = Check.NotNullOrWhiteSpace(title, nameof(title));

            Text = text;
            AssignedUserId = assignedUserId;

            Labels = new Collection<IssueLabel>();
        }

        private Issue() /* for deserialization & ORMs */
    }
}
```

- La clase **Issue** fuerza correctamente la creación de una entidad válida obteniendo las propiedades mínimas requeridas en su constructor como parámetros.
  - El constructor valida las entradas (`Check.NotNullOrWhiteSpace(...)` lanza `ArgumentException` si el valor dado está vacío).
  - Inicializa las subcolecciones, de modo que no obtengas una excepción de referencia nula cuando intentas usar la colección **Etiquetas** después de crear el **Problema**.
  - El constructor también toma el **id** y lo pasa a la clase **base**. No generamos **Guid** dentro del constructor para poder delegar esta responsabilidad a otro servicio (ver [Generación de Guid](#)).
- 
- El constructor **privado** vacío es necesario para los ORM. Lo hicimos **privado** para evitar usarlo accidentalmente en nuestro propio código.

Ver las [Entidades](#) Documento para obtener más información sobre la creación de entidades con el marco ABP.

## Accesores y métodos de propiedades de entidad

El ejemplo anterior puede parecerle extraño. Por ejemplo, forzamos a pasar un **título no nulo** en el constructor. Sin embargo, el desarrollador puede luego establecer la propiedad **Título como nula** sin ningún control. Esto se debe a que el código de ejemplo anterior solo se centra en el constructor.

Si declaramos todas las propiedades con establecedores públicos (como en el ejemplo de la **clase Issue** anterior), no podemos forzar la validez e integridad de la entidad en su ciclo de vida. Por lo tanto:

- Utilice **un establecedor privado** para una propiedad cuando necesite realizar alguna **lógica** al configurar esa propiedad.
- Definir métodos públicos para manipular dichas propiedades.

Ejemplo: Métodos para cambiar las propiedades de forma controlada

```
using System;
using Volo.Abp;
using Volo.Abp.Domain.Entities;

namespace IssueTracking.Issues
{
    public class Issue : AggregateRoot<Guid>
    {
        public Guid RepositoryId { get; private set; } //Never changes
        public string Title { get; private set; } //Needs validation
        public string Text { get; set; } //No validation
        public Guid? AssignedUserId { get; set; } //No validation
        public bool IsClosed { get; private set; } //Should be changed with CloseReason
        public IssueCloseReason? CloseReason { get; private set; } //Should be changed with IsCl

        //...

        public void SetTitle(string title)
        {
            Title = Check.NotNullOrWhiteSpace(title, nameof(title));
        }

        public void Close(IssueCloseReason reason)
        {
            IsClosed = true;
            CloseReason = reason;
        }

        public void ReOpen()
        {
            IsClosed = false;
            CloseReason = null;
        }
    }
}
```

- El definidor de `RepositoryId` se hizo privado y no hay forma de cambiarlo después de crear un `problema` porque esto es lo que queremos en este dominio: un problema no se puede mover a otro repositorio.
- El establecedor de `títulos` se hizo privado y el método `SetTitle` se ha eliminado. creado si desea cambiarlo más tarde de forma controlada.
- `Text` y `AssignedUserId` tienen configuradores públicos, ya que no hay restricciones sobre ellos. Pueden ser nulos o cualquier otro valor. Creemos que no es necesario definir métodos separados para configurarlos. Si lo necesitamos más adelante, podemos agregar métodos y hacer que los configuradores sean privados. Los cambios importantes no son un problema en la capa de dominio, ya que la capa de dominio es un proyecto interno y no está expuesta a los clientes.
- `IsClosed` y `IssueCloseReason` son propiedades de par. Se definieron los métodos `Close` y `ReOpen` para cambiarlos juntos. De esta manera, evitamos cerrar un problema sin cualquier razón

## Lógica de negocios y excepciones en las entidades

Al implementar la validación y la lógica de negocios en las entidades, con frecuencia es necesario gestionar los casos excepcionales.

En estos casos;

- Crear excepciones específicas del dominio.
- Lanza estas excepciones en los métodos de entidad cuando necesario.

Ejemplo:

```
public class Issue : AggregateRoot<Guid>
{
    //...

    public bool IsLocked { get; private set; }
    public bool IsClosed { get; private set; }
    public IssueCloseReason? CloseReason { get; private set; }

    public void Close(IssueCloseReason reason)
    {
        IsClosed = true;
        CloseReason = reason;
    }

    public void ReOpen()
    {
        if (IsLocked)
        {
            throw new IssueStateException(
                "Can not open a locked issue! Unlock it first."
            );
        }

        IsClosed = false;
        CloseReason = null;
    }

    public void Lock()
    {
        if (!IsClosed)
        {
            throw new IssueStateException(
                "Can not open a locked issue! Unlock it first."
            );
        }

        IsLocked = true;
    }

    public void Unlock()
    {
        IsLocked = false;
    }
}
```

Aquí hay dos reglas de negocio:

- Un problema bloqueado no se puede volver a abrir.
- No puedes bloquear un problema abierto.

La clase `Issue` lanza una `IssueStateException` en estos casos para forzar las reglas comerciales:

```
using System;

namespace IssueTracking.Issues
{
    public class IssueStateException : Exception
    {
        public IssueStateException(string message)
            : base(message)
        {
        }
    }
}
```

Existen dos problemas potenciales al lanzar tales excepciones:

1. En caso de una excepción de este tipo, ¿debería el **usuario final** ver el mensaje de excepción (error)? Si es así, ¿cómo se **localiza** el mensaje de excepción? No se puede utilizar la **localización** sistema, porque no puede inyectar y usar **IStringLocalizer** en las entidades.
2. Para una aplicación web o API HTTP, ¿qué **código de estado HTTP** debe devolver al cliente?

Manejo de excepciones de ABP El sistema resuelve estos y otros problemas similares.

Ejemplo: Lanzar una excepción comercial con código

```
using Volo.Abp;

namespace IssueTracking.Issues
{
    public class IssueStateException : BusinessException
    {
        public IssueStateException(string code)
            : base(code)
        {
        }
    }
}
```

- La clase `IssueStateException` hereda la clase `BusinessException` . ABP devuelve el código de estado HTTP 403 (prohibido) de forma predeterminada (en lugar de 500: error interno del servidor) para las excepciones derivadas de `BusinessException`.
- El código se utiliza como clave en el archivo de recursos de localización. para encontrar el mensaje localizado.

Ahora, podemos cambiar el método `ReOpen` como se muestra a continuación:

```
public void ReOpen()
{
    if (IsLocked)
    {
        throw new IssueStateException("IssueTracking:CanNotOpenLockedIssue");
    }

    IsClosed = false;
    CloseReason = null;
}
```

Utilice constantes en lugar de cadenas mágicas.

Y agregue una entrada al recurso de localización como se muestra a continuación:

```
"IssueTracking:CanNotOpenLockedIssue": "Can not open a locked issue! Unlock it first."
```

- Cuando se lanza la excepción, ABP utiliza automáticamente

Este mensaje localizado (basado en el idioma actual) para mostrar al usuario final.

- El código de excepción

(IssueTracking:CanNotOpenLockedIssue aquí) también se envía al cliente, por lo que puede manejar el caso de error mediante programación.

Para este ejemplo, podría lanzar directamente

`BusinessException` en lugar de definir una `IssueStateException` especializada. El resultado será el mismo. Consulte el [documento de manejo de excepciones](#) para todos los detalles.

## Lógica de Negocios en Entidades que Requieren Información Externa Servicios

Es sencillo implementar una regla de negocio en un método de entidad cuando la lógica de negocio solo utiliza las propiedades de esa entidad.

¿Qué sucede si la lógica empresarial requiere **consultar la base de datos** o utilizar cualquier **servicio externo** que deba resolverse a partir de la **inyección de dependencia?** Sistema. Recuerde; ¡Las entidades no pueden inyectar servicios!

Hay dos formas comunes de implementar dicha lógica empresarial:

- Implementar la lógica empresarial en un método de entidad y **obtener dependencias externas como parámetros** del método.
- Crear un **servicio de dominio**.

Los servicios de dominio se explicarán más adelante, pero ahora veamos cómo se pueden implementar en la clase de entidad.

Ejemplo: Regla de negocio: No se pueden asignar más de 3 problemas abiertos a un usuario simultáneamente

```
public class Issue : AggregateRoot<Guid>
{
    //...
    public Guid? AssignedUserId { get; private set; }

    public async Task AssignToAsync(AppUser user, IUserIssueService userIssueService)
    {
        var openIssueCount = await userIssueService.GetOpenIssueCountAsync(user.Id);

        if (openIssueCount >= 3)
        {
            throw new BusinessException("IssueTracking:ConcurrentOpenIssueLimit");
        }

        AssignedUserId = user.Id;
    }

    public void CleanAssignment()
    {
        AssignedUserId = null;
    }
}
```

- El establecedor de la propiedad `AssignedUserId` se hizo privado. Por lo tanto, la única forma de cambiarlo es usar los métodos `AssignToAsync` y `CleanAssignment`.
- `AssignToAsync` obtiene una entidad `AppUser`. En realidad, solo utiliza el `user.Id`, por lo que podría obtener un valor `Guid`, como `userId`. Sin embargo, de esta manera se garantiza que el valor `Guid` sea el `Id` de un usuario existente y no un valor `Guid` aleatorio.
- `IUserIssueService` es un servicio arbitrario que se utiliza para obtener el recuento de problemas abiertos de un usuario. Es responsabilidad de la parte del código (que llama a `AssignToAsync`) resolver `IUserIssueService` y pasarlo aquí.

- `AssignToAsync` genera una excepción si la regla comercial no se cumple.
- Finalmente, si todo es correcto, se establece la propiedad `AssignedUserId`.

Este método garantiza perfectamente la aplicación de la lógica empresarial cuando se desea asignar un problema a un usuario. Sin embargo, tiene algunos problemas:

- Hace que la clase de entidad **dependa de un servicio externo**, lo que hace que la entidad sea **complicada**.
- Esto dificulta el uso de la entidad. El código que utiliza la entidad ahora debe inyectar `IUserIssueService` y pasarlo al método `AssignToAsync`.

Una forma alternativa de implementar esta lógica de negocios es introducir un **Servicio de Dominio**, que se explicará más adelante.

## Repositorios

Un repositorio es una interfaz tipo colección que utilizan las capas de dominio y aplicación para acceder al sistema de persistencia de datos (la base de datos) para leer y escribir los objetos de negocio, generalmente los agregados.

Los principios del Repositorio Común son:

- Definir una interfaz de repositorio en la capa de dominio (porque se utiliza en las capas de dominio y aplicación), implementar en la capa de infraestructura ( proyecto EntityFrameworkCore en la plantilla de inicio).
- No incluya lógica de negocios dentro de los repositorios.
- La interfaz del repositorio debe ser independiente del proveedor de la base de datos o del ORM. Por ejemplo, no devuelva un `DbSet` desde un método del repositorio. `DbSet` es un objeto proporcionado por EF Core.
- Crear repositorios para raíces agregadas, no para todas entidades. Porque a las entidades de subcolección (de un agregado) se debe acceder a través de la raíz del agregado.

## No incluir lógica de dominio en los repositorios

Si bien esta regla parece obvia al principio, es fácil filtrar la lógica empresarial en los repositorios.

Ejemplo: Obtener problemas inactivos de un repositorio

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Volo.Abp.Domain.Repositories;

namespace IssueTracking.Issues
{
    public interface IIssueRepository : IRepository<Issue, Guid>
    {
        Task<List<Issue>> GetInActiveIssuesAsync();
    }
}
```

`IIssueRepository` amplía la interfaz estándar `IRepository<...>` añadiendo un método `GetInActiveIssuesAsync`. Este repositorio funciona con una clase `Issue` como esta:

```
public class Issue : AggregateRoot<Guid>, IHasCreationTime
{
    public bool IsClosed { get; private set; }
    public Guid? AssignedUserId { get; private set; }
    public DateTime CreationTime { get; private set; }
    public DateTime? LastCommentTime { get; private set; }
    //...
}
```

(el código muestra solo las propiedades que necesitamos para este ejemplo)

La regla dice que el repositorio no debe conocer las reglas comerciales.

La pregunta aquí es "¿Qué es un problema inactivo? ¿Es una definición de regla de negocio?"

Veamos la implementación para entenderlo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using IssueTracking.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using Volo.Abp.Domain.Repositories.EntityFrameworkCore;
using Volo.Abp.EntityFrameworkCore;

namespace IssueTracking.Issues
{
    public class EfCoreIssueRepository :
        EfCoreRepository<IssueTrackingDbContext, Issue, Guid>,
        IIssueRepository
    {
        public EfCoreIssueRepository(
            IDbContextProvider<IssueTrackingDbContext> dbContextProvider)
            : base(dbContextProvider)
        {
        }

        public async Task<List<Issue>> GetInActiveIssuesAsync()
        {
            var daysAgo30 = DateTime.Now.Subtract(TimeSpan.FromDays(30));

            var dbSet = await GetDbSetAsync();
            return await dbSet.Where(i =>

                //Open
                !i.IsClosed &&

                //Assigned to nobody
                i.AssignedUserId == null &&

                //Created 30+ days ago
                i.CreationTime < daysAgo30 &&

                //No comment or the last comment was 30+ days ago
                (i.LastCommentTime == null || i.LastCommentTime < daysAgo30)

            ).ToListAsync();
        }
    }
}
```

(Se utilizó EF Core para la implementación. Consulte el [documento de integración de EF Core](#)) para aprender a crear repositorios personalizados con EF Core).

Cuando verificamos la implementación `GetInActiveIssuesAsync`, vemos una regla comercial que define un problema inactivo: el problema debe estar abierto, no estar asignado a nadie, haberse creado hace más de 30 días y no tener comentarios en los últimos 30 días.

Esta es una definición implícita de una regla de negocio que está oculta dentro de un método de repositorio. El problema surge cuando necesitamos reutilizar esta lógica de negocio.

Por ejemplo, supongamos que queremos agregar un método booleano `IsActive()` en la entidad `Issue`. De esta manera, podemos verificar la actividad cuando tenemos una entidad `Issue`.

Veamos la implementación:

```
public class Issue : AggregateRoot<Guid>, IHasCreationTime
{
    public bool IsClosed { get; private set; }
    public Guid? AssignedUserId { get; private set; }
    public DateTime CreationTime { get; private set; }
    public DateTime? LastCommentTime { get; private set; }
    //...

    public bool IsActive()
    {
        var daysAgo30 = DateTime.Now.Subtract(TimeSpan.FromDays(30));
        return
            //Open
            !IsClosed &&
            //Assigned to nobody
            AssignedUserId == null &&
            //Created 30+ days ago
            CreationTime < daysAgo30 &&
            //No comment or the last comment was 30+ days ago
            (LastCommentTime == null || LastCommentTime < daysAgo30);
    }
}
```

Tuvimos que copiar/pegar/modificar el código. ¿Qué pasa si cambia la definición de la actividad? No debemos olvidarnos de actualizar ambos lugares. Esto es una duplicación de la lógica empresarial, lo cual es bastante peligroso.

¡Una buena solución a este problema es el Patrón de Especificación!

## Presupuesto

Una especificación es una clase con nombre, reutilizable, combinable y comprobable para filtrar los objetos de dominio según las reglas de negocio.

ABP Framework proporciona la infraestructura necesaria para crear fácilmente clases de especificación y usarlas dentro del código de la aplicación. Implementemos el filtro de problemas inactivos como una clase de especificación:

```
using System;
using System.Linq.Expressions;
using Volo.Abp.Specifications;

namespace IssueTracking.Issues
{
    public class InActiveIssueSpecification : Specification<Issue>
    {
        public override Expression<Func<Issue, bool>> ToExpression()
        {
            var daysAgo30 = DateTime.Now.Subtract(TimeSpan.FromDays(30));
            return i =>

                //Open
                !i.IsClosed &&

                //Assigned to nobody
                i.AssignedUserId == null &&

                //Created 30+ days ago
                i.CreationTime < daysAgo30 &&

                //No comment or the last comment was 30+ days ago
                (i.LastCommentTime == null || i.LastCommentTime < daysAgo30);
        }
    }
}
```

La clase base `Specification<T>` simplifica la creación de una clase de especificación mediante la definición de una expresión. Simplemente moví la expresión aquí, desde el repositorio.

Ahora, podemos reutilizar `InActiveIssueSpecification` en la entidad `Issue` y en las clases `EfCoreIssueRepository`.

## Utilizando dentro de la Entidad

La clase `de especificación` proporciona un método `IsSatisfiedBy` que devuelve `verdadero` si el objeto (entidad) dado satisface la especificación. Podemos reescribir el método `Issue.IsActive` como se muestra a continuación:

```
public class Issue : AggregateRoot<Guid>, IHasCreationTime
{
    public bool IsClosed { get; private set; }
    public Guid? AssignedUserId { get; private set; }
    public DateTime CreationTime { get; private set; }
    public DateTime? LastCommentTime { get; private set; }
    //...

    public bool IsActive()
    {
        return new InActiveIssueSpecification().IsSatisfiedBy(this);
    }
}
```

Acabo de crear una nueva instancia de `InActiveIssueSpecification` y utilizó su método `IsSatisfiedBy` para reutilizar la expresión definido por la especificación.

## Uso con los repositorios

Primero, comenzando desde la interfaz del repositorio:

```
public interface IIssueRepository : IRepository<Issue, Guid>
{
    Task<List<Issue>> GetIssuesAsync(ISpecification<Issue> spec);
}
```

Se cambió el nombre de `GetInActiveIssuesAsync` a `GetIssuesAsync` simple al tomar un objeto de especificación. Dado que la especificación (el filtro) se ha sacado del repositorio, ya no necesitamos crear métodos diferentes para obtener problemas con diferentes condiciones (como `GetAssignedIssues(...)`, `GetLockedIssues(...)`, etc.)

La implementación actualizada del repositorio puede ser así:

```
public class EfCoreIssueRepository :
    EfCoreRepository<IssueTrackingDbContext, Issue, Guid>,
    IIssueRepository
{
    public EfCoreIssueRepository(
        IDbContextProvider<IssueTrackingDbContext> dbContextProvider)
        : base(dbContextProvider)
    {
    }

    public async Task<List<Issue>> GetIssuesAsync(ISpecification<Issue> spec)
    {
        var dbSet = await GetDbSetAsync();
        return await dbSet
            .Where(spec.ToExpression())
            .ToListAsync();
    }
}
```

Dado que el método `ToExpression()` devuelve una expresión, se puede pasar directamente al método `Where` para filtrar las entidades.

Finalmente, podemos pasar cualquier instancia de Especificación a la Método `GetIssuesAsync`:

```
public class IssueAppService : ApplicationService, IIssueAppService
{
    private readonly IIssueRepository _issueRepository;

    public IssueAppService(IIssueRepository issueRepository)
    {
        _issueRepository = issueRepository;
    }

    public async Task DoItAsync()
    {
        var issues = await _issueRepository.GetIssuesAsync(
            new InActiveIssueSpecification()
        );
    }
}
```

## Con repositorios predeterminados

En realidad, no es necesario crear repositorios personalizados para poder utilizar especificaciones. El `IRepository` estándar ya extiende el `IQueryable`, por lo que puede utilizar los métodos de extensión LINQ estándar sobre él:

```
public class IssueAppService : ApplicationService, IIssueAppService
{
    private readonly IRepository<Issue, Guid> _issueRepository;

    public IssueAppService(IRepository<Issue, Guid> issueRepository)
    {
        _issueRepository = issueRepository;
    }

    public async Task DoItAsync()
    {
        var queryable = await _issueRepository.GetQueryableAsync();
        var issues = AsyncExecuter.ToListAsync(
            queryable.Where(new InActiveIssueSpecification()))
        );
    }
}
```

[AsyncExecuter](#) es una utilidad proporcionada por ABP Framework para utilizar métodos de extensión LINQ asincrónicos (como [ToListAsync](#) aquí) sin depender del paquete NuGet de EF Core. Consulte el [documento Repositorios](#) Para más información.

## Combinando las especificaciones

Un aspecto poderoso de las Especificaciones es que son combinables. Supongamos que tenemos otra especificación que devuelve [verdadero](#) Sólo si el [problema](#) está en un hito:

```
public class MilestoneSpecification : Specification<Issue>
{
    public Guid MilestoneId { get; }

    public MilestoneSpecification(Guid milestoneId)
    {
        MilestoneId = milestoneId;
    }

    public override Expression<Func<Issue, bool>> ToExpression()
    {
        return i => i.MilestoneId == MilestoneId;
    }
}
```

Esta especificación es paramétrica , a diferencia de la [especificación InActiveIssueSpecification](#). Podemos combinar ambas especificaciones para obtener una lista de problemas inactivos en un hito específico:

```
public class IssueAppService : ApplicationService, IIssueAppService
{
    private readonly IRepository<Issue, Guid> _issueRepository;

    public IssueAppService(IRepository<Issue, Guid> issueRepository)
    {
        _issueRepository = issueRepository;
    }

    public async Task DoItAsync(Guid milestoneId)
    {
        var queryable = await _issueRepository.GetQueryableAsync();
        var issues = AsyncExecuter.ToListAsync(
            queryable
                .Where(
                    new InActiveIssueSpecification()
                        .And(new MilestoneSpecification(milestoneId))
                        .ToExpression()
                )
        );
    }
}
```

El ejemplo anterior utiliza el **método de extensión And** para combinar las especificaciones. Hay más métodos de combinación disponibles, como **Or(...)** y **AndNot(...)**.

Ver el [documento de Especificaciones](#) para obtener más detalles sobre la infraestructura de especificación proporcionada por el Marco ABP.

## Servicios de dominio

Los servicios de dominio implementan la lógica de dominio que;

- Depende de **los servicios y repositorios**.
- Necesita trabajar con **múltiples agregados**, por lo que la lógica no encaja correctamente en ninguno de ellos.

Los servicios de dominio funcionan con objetos de dominio. Sus métodos pueden **obtener** y **devolver entidades, objetos de valor, tipos primitivos, etc.**

Sin embargo, **no obtienen ni devuelven DTO**. Los DTO son parte de la capa de aplicación.

## Ejemplo: Asignar un problema a un usuario

Recuerde cómo se ha implementado una asignación de problema en la entidad **Problema** :

```
public class Issue : AggregateRoot<Guid>
{
    //...
    public Guid? AssignedUserId { get; private set; }

    public async Task AssignToAsync(AppUser user, IUserIssueService userIssueService)
    {
        var openIssueCount = await userIssueService.GetOpenIssueCountAsync(user.Id);

        if (openIssueCount >= 3)
        {
            throw new BusinessException("IssueTracking:ConcurrentOpenIssueLimit");
        }

        AssignedUserId = user.Id;
    }

    public void CleanAssignment()
    {
        AssignedUserId = null;
    }
}
```

Aquí trasladaremos esta lógica a un servicio de dominio.

Primero, cambiando la clase **Issue** :

```
public class Issue : AggregateRoot<Guid>
{
    //...
    public Guid? AssignedUserId { get; internal set; }
}
```

- Se eliminaron los métodos relacionados con la asignación.
- Se cambió el establecedor de la propiedad `AssignedUserId` de privado a interno, para permitir configurarlo desde el Servicio de Dominio.

El siguiente paso es crear un servicio de dominio, llamado `IssueManager`, que tenga `AssignToAsync` para asignar el problema dado al usuario dado.

```
public class IssueManager : DomainService
{
    private readonly IRepository<Issue, Guid> _issueRepository;

    public IssueManager(IRepository<Issue, Guid> issueRepository)
    {
        _issueRepository = issueRepository;
    }

    public async Task AssignToAsync(Issue issue, AppUser user)
    {
        var openIssueCount = await _issueRepository.CountAsync(
            i => i.AssignedUserId == user.Id && !i.IsClosed
        );

        if (openIssueCount >= 3)
        {
            throw new BusinessException("IssueTracking:ConcurrentOpenIssueLimit");
        }

        issue.AssignedUserId = user.Id;
    }
}
```

`IssueManager` puede inyectar cualquier dependencia de servicio y usarla para consultar el recuento de problemas abiertos del usuario.

Preferimos y sugerimos utilizar el sufijo **Manager** para los Servicios de Dominio.

El único problema de este diseño es que **Issue.AssignedUserId** ahora está abierto para configurarse fuera de la clase. Sin embargo, no es **público**. Es **interno** y solo es posible cambiarlo dentro del mismo ensamblaje, el proyecto **IssueTracking.Domain** para esta solución de ejemplo. Creemos que esto es razonable;

- Los desarrolladores de la capa de dominio ya conocen las reglas del dominio y utilizan **IssueManager**.
- Los desarrolladores de la capa de aplicación ya están obligados a utilizar **IssueManager** ya que no lo configuran directamente.

Si bien existe un equilibrio entre dos enfoques, preferimos crear servicios de dominio cuando la lógica empresarial requiere trabajar con servicios externos.

Si no tiene una buena razón, creemos que no es necesario crear interfaces (como **IIssueManager** para **IssueManager**) para los servicios de dominio.

## Servicios de aplicación

Un servicio de aplicación es un servicio sin estado que implementa casos de uso de la aplicación. Un servicio de aplicación normalmente obtiene y devuelve DTO. Lo utiliza la capa de presentación. Utiliza y coordina los objetos de dominio (entidades, repositorios, etc.) para implementar los casos de uso.

Los principios comunes de un servicio de aplicación son:

- Implemente la lógica de la aplicación específica para el caso de uso actual. No implemente la lógica del dominio central dentro de los servicios de la aplicación. Volveremos a hablar de las diferencias entre las lógicas del dominio de la aplicación.
- Nunca obtenga ni devuelva entidades para un método de servicio de aplicación. Esto rompe la encapsulación de la capa de dominio. Siempre obtenga y devuelva DTO.

## Ejemplo: Asignar un problema a un usuario

```
using System;
using System.Threading.Tasks;
using IssueTracking.Users;
using Microsoft.AspNetCore.Authorization;
using Volo.Abp.Application.Services;
using Volo.Abp.Domain.Repositories;

namespace IssueTracking.Issues
{
    public class IssueAppService : ApplicationService, IIssueAppService
    {
        private readonly IssueManager _issueManager;
        private readonly IRepository<Issue, Guid> _issueRepository;
        private readonly IRepository<AppUser, Guid> _userRepository;

        public IssueAppService(
            IssueManager issueManager,
            IRepository<Issue, Guid> issueRepository,
            IRepository<AppUser, Guid> userRepository)
        {
            _issueManager = issueManager;
            _issueRepository = issueRepository;
            _userRepository = userRepository;
        }

        [Authorize]
        public async Task AssignAsync(IssueAssignDto input)
        {
            var issue = await _issueRepository.GetAsync(input.IssueId);
            var user = await _userRepository.GetAsync(input.UserId);

            await _issueManager.AssignToAsync(issue, user);

            await _issueRepository.UpdateAsync(issue);
        }
    }
}
```

Un método de servicio de aplicación normalmente tiene tres pasos que se implementan aquí:

1. Obtenga los objetos de dominio relacionados de la base de datos a implementar el caso de uso.
2. Utilice objetos de dominio (servicios de dominio, entidades, etc.) para realizar la operación real.
3. Actualice las entidades modificadas en la base de datos.

`IssueAssignDto` en este ejemplo es una clase DTO simple:

La última actualización no es necesaria si está utilizando EF Core, ya que cuenta con un sistema de seguimiento de cambios. Si desea aprovechar esta característica de EF Core, consulte la sección Discusión sobre el principio de independencia de la base de datos que aparece más arriba.

`IssueAssignDto` en este ejemplo es una clase DTO simple:

```
using System;

namespace IssueTracking.Issues
{
    public class IssueAssignDto
    {
        public Guid IssueId { get; set; }
        public Guid UserId { get; set; }
    }
}
```

## Transferencia de datos Objetos

Una **DTO** es un objeto simple que se utiliza para transferir estado (datos) entre las capas de aplicación y presentación. Por lo tanto, los métodos del servicio de aplicación obtienen y devuelven DTO.

### Principios y mejores prácticas comunes de las DTO

- Un DTO **debe ser serializable** por naturaleza, ya que la mayoría de las veces se transfiere a través de la red. Por lo tanto, debe tener un constructor sin parámetros (vacío).
- No debe contener ninguna lógica comercial.
- Nunca heredar ni hacer referencia a **entidades**.

Los **DTO de entrada** (aquellos que se pasan a los métodos del servicio de aplicación) tienen una naturaleza diferente a la de **los DTO de salida** (aquellos que se devuelven desde los métodos del servicio de aplicación), por lo que se tratarán de forma diferente.

## Mejores prácticas para la entrada de DTO

### No defina propiedades no utilizadas para los DTO de entrada

Defina únicamente las propiedades necesarias para el caso de uso. De lo contrario, será confuso para los clientes utilizar el método del servicio de aplicación.

Seguramente puede definir propiedades opcionales, pero deberían afectar el funcionamiento del caso de uso cuando el cliente las proporciona.

Esta regla parece innecesaria en primer lugar. ¿Quién definiría parámetros no utilizados (propiedades DTO de entrada) para un método? Pero sucede, especialmente cuando se intenta reutilizar DTO de entrada.

### No reutilice los DTO de entrada

Defina un DTO de entrada especializado para cada caso de uso (método de servicio de aplicación). De lo contrario, algunas propiedades no se utilizan en algunos casos y esto viola la regla definida anteriormente: No

Definir propiedades no utilizadas para los DTO de entrada.

A veces, parece atractivo reutilizar la misma clase DTO para dos casos de uso, porque son casi iguales. Incluso si ahora son iguales, probablemente se volverán diferentes con el tiempo y se llegará al mismo problema. La duplicación de código es una mejor práctica que acoplar casos de uso.

Otra forma de reutilizar los DTO de entrada es heredándolos entre sí. Si bien esto puede ser útil en algunos casos excepcionales, la mayoría de las veces conduce al mismo punto.

Ejemplo: Servicio de aplicación de usuario

```
public interface IUserAppService : IApplicationService
{
    Task CreateAsync(UserDto input);
    Task UpdateAsync(UserDto input);
    Task ChangePasswordAsync(UserDto input);
}
```

IUserAppService utiliza UserDto como DTO de entrada en todos los métodos (casos de uso). UserDto se define a continuación:

```
public class UserDto
{
    public Guid Id { get; set; }
    public string UserName { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }
    public DateTime CreationTime { get; set; }
}
```

Para este ejemplo;

- No se utiliza Id en Create ya que lo determina el servidor.
- La contraseña no se utiliza en la actualización ya que tenemos otro método para ello.
- CreationTime nunca se utiliza ya que no podemos permitir que el cliente Envía la hora de creación. Debe estar configurada en el servidor.

Una verdadera implementación podría ser así:

```
public interface IUserAppService : IApplicationService
{
    Task CreateAsync(UserCreationDto input);
    Task UpdateAsync(UserUpdateDto input);
    Task ChangePasswordAsync(UserChangePasswordDto input);
}
```

Con las clases DTO de entrada dadas:

```
public class UserCreationDto
{
    public string UserName { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }
}

public class UserUpdateDto
{
    public Guid Id { get; set; }
    public string UserName { get; set; }
    public string Email { get; set; }
}

public class UserChangePasswordDto
{
    public Guid Id { get; set; }
    public string Password { get; set; }
}
```

Este es un enfoque más fácil de mantener aunque se escribe más código.

**Caso excepcional:** puede haber algunas excepciones para esta regla: si siempre desea desarrollar dos métodos **en paralelo**, es posible que comparten el mismo DTO de entrada (por herencia o reutilización directa). Por ejemplo, si tiene una página de informes que tiene algunos filtros y tiene varios métodos de servicio de aplicación (como métodos de informe de pantalla, informe de Excel y informe CSV) que usan los mismos filtros pero devuelven resultados diferentes, es posible que desee reutilizar el mismo DTO de entrada de filtro para **acoplar estos casos de uso**. Porque, en este ejemplo, siempre que cambia un filtro, tiene que realizar los cambios necesarios en todos los métodos para tener un sistema de informes consistente.

## Lógica de validación de DTO de entrada

- Implemente solo **la validación formal** dentro del DTO. Utilice atributos de validación de anotación de datos o implemente **IValidatableObject** para la validación formal.
- No realice **la validación del dominio**. Por ejemplo, no intente comprobar la restricción de nombre de usuario único en los DTO.

## Ejemplo: Uso de atributos de anotación de datos

```
using System.ComponentModel.DataAnnotations;

namespace IssueTracking.Users
{
    public class UserCreationDto
    {
        [Required]
        [StringLength(UserConsts.MaxUserNameLength)]
        public string UserName { get; set; }

        [Required]
        [EmailAddress]
        [StringLength(UserConsts.MaxEmailLength)]
        public string Email { get; set; }

        [Required]
        [StringLength(UserConsts.MaxEmailLength,
        MinimumLength = UserConsts.MinPasswordLength)]
        public string Password { get; set; }
    }
}
```

ABP Framework valida automáticamente los DTO de entrada y arroja `AbpValidationException` y devuelve el estado HTTP `400` al cliente en caso de una entrada no válida.

Algunos desarrolladores creen que es mejor separar las reglas de validación y las clases DTO. Creemos que el enfoque declarativo (anotación de datos) es práctico y útil y no causa ningún problema de diseño. Sin embargo, ABP también admite [la integración de FluentValidation](#). Si prefieres el otro enfoque.

Ver el [documento de Validación](#) para todas las opciones de validación.

## Mejores prácticas de DTO de salida

- Mantenga el recuento de DTO de salida al mínimo. Reutilice siempre que sea posible (excepción: no reutilice los DTO de entrada como DTO de salida).
- Los DTO de salida pueden contener más propiedades que las utilizadas en el código del cliente.
- Devolver la entidad DTO desde los métodos Crear y Actualizar .

Los principales objetivos de estas sugerencias son:

- Hacer que el código del cliente sea fácil de desarrollar y ampliar;
  - Tratar con **DTO similares, pero no iguales**, es problemático para el cliente.
  - Es común que **en el futuro se necesiten otras propiedades** en la interfaz de usuario o el cliente. Devolver todas las propiedades (teniendo en cuenta la seguridad y los privilegios) de una entidad facilita la mejora del código del cliente sin necesidad de tocar el código del backend.
  - Si abre su API a **clientes de terceros** y no conoce los requisitos de cada uno de ellos,
  - Haga que el código del lado del servidor sea fácil de desarrollar y ampliar;
  - Tiene menos clases que comprender y mantener.
  - Puede reutilizar la asignación de objetos Entidad->DTO código.
- Devolver los mismos tipos desde diferentes métodos hace que sea fácil y claro crear **nuevos métodos**.

## Ejemplo: Devolver diferentes DTO desde diferentes métodos

```
public interface IUserAppService : IApplicationService
{
    UserDto Get(Guid id);
    List<UserNameAndEmailDto> GetUserNameAndEmail(Guid id);
    List<string> GetRoles(Guid id);
    List<UserListDto> GetList();
    UserCreateResultDto Create(UserCreationDto input);
    UserUpdateResultDto Update(UserUpdateDto input);
}
```

(No usamos métodos asíncronos para hacer el ejemplo más claro,  
¡pero usa async en tu aplicación del mundo real!)

El código de ejemplo anterior devuelve distintos tipos de DTO para cada método. Como puede suponer, habrá muchas duplicaciones de código para consultar datos y asignar entidades a DTO.

El servicio **IUserAppService** anterior se puede simplificar:

```
public interface IUserAppService : IApplicationService
{
    UserDto Get(Guid id);
    List<UserDto> GetList();
    UserDto Create(UserCreationDto input);
    UserDto Update(UserUpdateDto input);
}
```

Con un DTO de salida única:

```
public class UserDto
{
    public Guid Id { get; set; }
    public string UserName { get; set; }
    public string Email { get; set; }
    public DateTime CreationTime { get; set; }
    public List<string> Roles { get; set; }
}
```

- Se eliminaron `GetUserNameAndEmail` y `GetRoles` desde `Get`

El método ya devuelve la información necesaria.

- `GetList` ahora devuelve lo mismo que `Get`.

- `Crear y actualizar` también devuelve el mismo `UserDto`.

El uso del mismo DTO tiene muchas ventajas, como se explicó anteriormente.

Por ejemplo, piense en un escenario en el que muestra una **cuadrícula de datos** de usuarios en la interfaz de usuario. Después de actualizar un usuario, puede obtener el valor de retorno y **actualizarlo en la interfaz de usuario**. Por lo tanto, no necesita llamar a `GetList` nuevamente. Es por eso que sugerimos devolver la entidad DTO (`UserDto` aquí) como valor de retorno de las operaciones **de creación y actualización**.

## Discusión

Es posible que algunas de las sugerencias de DTO de salida no se adapten a todos los escenarios. Estas sugerencias se pueden ignorar por razones de rendimiento , especialmente cuando se devuelven grandes conjuntos de datos o cuando crea servicios para su propia interfaz de usuario y tiene demasiadas solicitudes simultáneas.

En estos casos, es posible que desee crear DTO de salida especializados con información mínima. Las sugerencias anteriores son especialmente para aplicaciones en las que mantener la base de código es más importante que una pérdida insignificante de rendimiento.

## Mapeo de objeto a objeto

Mapeo automático [de objeto a objeto](#) es un enfoque útil para copiar valores de un objeto a otro cuando dos objetos tienen propiedades iguales o similares.

Las clases DTO y Entity generalmente tienen propiedades iguales o similares y normalmente es necesario crear objetos DTO a partir de Entities.

[Sistema de mapeo de objeto a objeto de ABP con AutoMapper](#) La integración hace que estas operaciones sean mucho más fáciles en comparación con el mapeo manual.

- Utilice el mapeo automático de objetos solo para que la entidad genere mapeos DTO.
- No utilice el mapeo automático de objetos para las asignaciones de DTO de entrada a entidad .

Existen algunas razones por las que no debería utilizar DTO de entrada para Mapeo automático de entidades;

1. Una clase de entidad normalmente tiene un **constructor** que toma parámetros y garantiza la creación de objetos válidos. La operación de mapeo automático de objetos generalmente requiere un espacio vacío constructor.
2. La mayoría de las propiedades de la entidad tendrán **configuradores privados** y usted debe usar métodos para cambiar estas propiedades de manera controlada.
3. Por lo general, es necesario **validar y procesar cuidadosamente** los entrada del usuario/cliente en lugar de asignar ciegamente las propiedades de la entidad.

Si bien algunos de estos problemas se pueden resolver mediante configuraciones de mapeo (por ejemplo, AutoMapper permite definir reglas de mapeo personalizadas), esto hace que el código de su negocio **sea implícito/oculto y esté estrechamente vinculado a la infraestructura**. Creemos que el código de negocio debe ser explícito, claro y fácil de entender.

Consulte la **sección Creación de entidades** a continuación para ver un ejemplo de implementación de las sugerencias realizadas en esta sección.

# Ejemplos de casos de uso

En esta sección se mostrarán algunos casos de uso de ejemplo y se analizarán escenarios alternativos.

## Creación de entidades

La creación de un objeto a partir de una clase Entity/Aggregate Root es el primer paso del ciclo de vida de esa entidad. La sección [Aggregate/Aggregate Root Rules & Best Practices](#) sugiere **crear un constructor principal** para la clase Entity que garantice la **creación de una entidad válida**. Por lo tanto, siempre que necesitemos crear una instancia de esa entidad, debemos **usar ese constructor**.

Vea la [clase de raíz agregada del problema](#) a continuación:

```
public class Issue : AggregateRoot<Guid>
{
    public Guid RepositoryId { get; private set; }
    public string Title { get; private set; }
    public string Text { get; set; }
    public Guid? AssignedUserId { get; internal set; }

    public Issue(
        Guid id,
        Guid repositoryId,
        string title,
        string text = null
    ) : base(id)
    {
        RepositoryId = repositoryId;
        Title = Check.NotNullOrEmptyWhiteSpace(title, nameof(title));
        Text = text; //Allow empty/null
    }

    private Issue() { /* Empty constructor is for ORMs */ }

    public void SetTitle(string title)
    {
        Title = Check.NotNullOrEmptyWhiteSpace(title, nameof(title));
    }

    //...
}
```

- Esta clase garantiza la creación de una entidad válida por su constructor.
- Si necesita cambiar el **título** más tarde, deberá utilizar el **método SetTitle** que continúa manteniendo **el título** en un estado válido.
- Si desea asignar este problema a un usuario, debe utilizar **IssueManager** (implementa algunas reglas comerciales antes de la asignación; consulte la **sección Servicios de dominio** más arriba para recordar).

- La propiedad **Texto** tiene un establecedor público, porque también acepta valores nulos y no tiene reglas de validación para este ejemplo. También es opcional en el constructor.

Veamos un método de servicio de aplicación que se utiliza para crear un problema:

```
public class IssueAppService : ApplicationService, IIssueAppService
{
    private readonly IssueManager _issueManager;
    private readonly IRepository<Issue, Guid> _issueRepository;
    private readonly IRepository<AppUser, Guid> _userRepository;

    public IssueAppService(
        IssueManager issueManager,
        IRepository<Issue, Guid> issueRepository,
        IRepository<AppUser, Guid> userRepository)
    {
        _issueManager = issueManager;
        _issueRepository = issueRepository;
        _userRepository = userRepository;
    }

    public async Task<IssueDto> CreateAsync(IssueCreationDto input)
    {
        // Create a valid entity
        var issue = new Issue(
            GuidGenerator.Create(),
            input.RepositoryId,
            input.Title,
            input.Text
        );

        // Apply additional domain actions
        if (input.AssignedUserId.HasValue)
        {
            var user = await _userRepository.GetAsync(input.AssignedUserId.Value);
            await _issueManager.AssignToAsync(issue, user);
        }

        // Save
        await _issueRepository.InsertAsync(issue);

        // Return a DTO represents the new Issue
        return ObjectMapper.Map<Issue, IssueDto>(issue);
    }
}
```

## Método `CreateAsync` ;

- Utiliza el constructor `Issue` para crear un problema válido. pasa el `Id` usando `IGuidGenerator` Servicio. Aquí no se utiliza el mapeo automático de objetos.
  - Si el cliente desea asignar este problema a un usuario al crear un objeto, utiliza `IssueManager` para hacerlo, permitiéndole realizar las comprobaciones necesarias antes de esta asignación.
- 
- Guarda la entidad en la base de datos.
  - Finalmente, utiliza `IObjectMapper` para devolver un `IssueDto` que se crea automáticamente mediante el mapeo desde la nueva entidad `Issue` .

## Aplicación de reglas de dominio en la creación de entidades

La entidad de ejemplo `Issue` no tiene ninguna regla de negocio sobre la creación de entidades, excepto algunas validaciones formales en el constructor. Sin embargo, puede haber escenarios en los que la creación de entidades deba verificar algunas reglas comerciales adicionales.

Por ejemplo, supongamos que no desea permitir la creación de un problema si ya existe un problema con exactamente el mismo título. ¿Dónde implementar esta regla? No es adecuado implementar esta regla en el servicio de aplicaciones, ya que es una regla central del negocio (dominio) que siempre debe verificarse.

Esta regla se debe implementar en un servicio de dominio, `IssueManager` en este caso. Por lo tanto, debemos obligar a la capa de aplicación a utilizar siempre `IssueManager` para crear un nuevo problema.

Primero, podemos hacer que el constructor de `Issue` sea interno, en lugar de público:

```
public class Issue : AggregateRoot<Guid>
{
    //...

    internal Issue(
        Guid id,
        Guid repositoryId,
        string title,
        string text = null
    ) : base(id)
    {
        RepositoryId = repositoryId;
        Title = Check.NotNullOrWhiteSpace(title, nameof(title));
        Text = text; //Allow empty/null
    }

    //...
}
```

Esto evita que los servicios de aplicación utilicen directamente el constructor, por lo que utilizarán `IssueManager`. Luego, podemos agregar un método `CreateAsync` a `IssueManager`:

```
using System;
using System.Threading.Tasks;
using Volo.Abp;
using Volo.Abp.Domain.Repositories;
using Volo.Abp.Domain.Services;

namespace IssueTracking.Issues
{
    public class IssueManager : DomainService
    {
        private readonly IRepository<Issue, Guid> _issueRepository;

        public IssueManager(IRepository<Issue, Guid> issueRepository)
        {
            _issueRepository = issueRepository;
        }

        public async Task<Issue> CreateAsync(
            Guid repositoryId,
            string title,
            string text = null)
        {
            if (await _issueRepository.AnyAsync(i => i.Title == title))
            {
                throw new BusinessException("IssueTracking:IssueWithSameTitleExists");
            }

            return new Issue(
                GuidGenerator.Create(),
                repositoryId,
                title,
                text
            );
        }
    }
}
```

- El método **CreateAsync** verifica si ya existe un problema con el mismo título y genera una excepción comercial en este caso.
- Si no hay duplicación, crea y devuelve un nuevo **problema**.

El `IssueAppService` se modifica como se muestra a continuación para utilizar el método `CreateAsync` de `IssueManager`:

```

public class IssueAppService : ApplicationService, IIssueAppService
{
    private readonly IssueManager _issueManager;
    private readonly IRepository<Issue, Guid> _issueRepository;
    private readonly IRepository<AppUser, Guid> _userRepository;

    public IssueAppService(
        IssueManager issueManager,
        IRepository<Issue, Guid> issueRepository,
        IRepository<AppUser, Guid> userRepository)
    {
        _issueManager = issueManager;
        _issueRepository = issueRepository;
        _userRepository = userRepository;
    }

    public async Task<IssueDto> CreateAsync(IssueCreationDto input)
    {
        // Create a valid entity using the IssueManager
        var issue = await _issueManager.CreateAsync(
            input.RepositoryId,
            input.Title,
            input.Text
        );

        // Apply additional domain actions
        if (input.AssignedUserId.HasValue)
        {
            var user = await _userRepository.GetAsync(input.AssignedUserId.Value);
            await _issueManager.AssignToAsync(issue, user);
        }

        // Save
        await _issueRepository.InsertAsync(issue);

        // Return a DTO represents the new Issue
        return ObjectMapper.Map<Issue, IssueDto>(issue);
    }
}

// *** IssueCreationDto class ***
public class IssueCreationDto
{
    public Guid RepositoryId { get; set; }
    [Required]
    public string Title { get; set; }
    public Guid? AssignedUserId { get; set; }
    public string Text { get; set; }
}

```

## Discusión: ¿Por qué el problema no se guarda en la base de datos en IssueManager?

Puede que se pregunte "¿Por qué IssueManager no guardó el problema en la base de datos?". Creemos que es responsabilidad del servicio de aplicaciones.

Debido a que el servicio de aplicación puede requerir cambios u operaciones adicionales en el objeto Issue antes de guardarlo.

Si el servicio de dominio lo guarda, la operación de guardado se duplica.

- Provoca pérdida de rendimiento debido al doble viaje de ida y vuelta a la base de datos.
- Requiere una transacción de base de datos explícita que cubra ambas operaciones.
- Si acciones adicionales cancelan la creación de la entidad debido a una regla comercial, la transacción debe revertirse en la base de datos.

Cuando verifique IssueAppService, verá la ventaja de no guardar el Issue en la base de datos en IssueManager.CreateAsync. De lo contrario, tendríamos que realizar una inserción (en IssueManager) y una actualización (después de la asignación).

## Discusión: ¿Por qué la verificación de título duplicado no está implementada en el servicio de aplicación?

Podríamos decir simplemente "Porque es una lógica de dominio central y debería implementarse en la capa de dominio". Sin embargo, esto plantea una nueva pregunta : "¿Cómo decidió que es una lógica de dominio central, pero no una lógica de aplicación?" (Discutiremos la diferencia más adelante con más detalles).

En este ejemplo, una pregunta sencilla puede ayudarnos a tomar la decisión: "Si tenemos otra forma (caso de uso) de crear un problema, ¿deberíamos aplicar la misma regla? ¿Esa regla debería implementarse siempre ?". Puede que pienses "¿Por qué tenemos una segunda forma de crear un problema?". Sin embargo, en la vida real, tienes:

- Los **usuarios finales** de la aplicación pueden crear problemas en la interfaz de usuario estándar de su aplicación.
- Es posible que tenga una segunda **aplicación de back office** que utilicen sus propios empleados y que desee proporcionar una forma de crear problemas (probablemente con diferentes reglas de autorización en este caso).
- Es posible que tenga una API HTTP que esté abierta a **terceros**. **clientes** y crean problemas.
- Es posible que tenga un servicio **de trabajo en segundo plano** que lo haga algo y crea problemas si detecta algunos problemas.  
De esta forma, se creará un problema sin ninguna interacción del usuario (y probablemente sin ninguna verificación de autorización estándar).

- Es posible que tenga un botón en la interfaz de usuario que **convierta** algo (por ejemplo, una discusión) sobre un tema.

Podemos dar más ejemplos. Todos ellos deben implementarse mediante diferentes **métodos de servicio de aplicación** (consulte la sección Capas de aplicación múltiples a continuación), pero siempre deben seguir la regla: ¡el título del nuevo problema no puede ser el mismo que el de ningún problema existente! Es por eso que esta lógica es una **lógica de dominio central**, debe ubicarse en la capa de dominio y **no debe duplicarse** en todos estos métodos de servicio de aplicación.

## Actualizar/manipular una entidad

Una vez que se crea una entidad, los casos de uso la actualizan o manipulan hasta que se elimina del sistema. Puede haber distintos tipos de casos de uso que modifiquen una entidad de forma directa o indirecta.

En esta sección, analizaremos una operación de actualización típica que cambia múltiples propiedades de un **problema**.

Esta vez, comenzando desde la Actualización DTO:

```
public class UpdateIssueDto
{
    [Required]
    public string Title { get; set; }
    public string Text { get; set; }
    public Guid? AssignedUserId { get; set; }
}
```

Al comparar con `IssueCreationDto`, no ves ningún `RepositoryId`.

Porque nuestro sistema no permite mover problemas entre repositorios (como los repositorios de GitHub). Solo `se requiere el título` y las demás propiedades son opcionales.

Veamos la implementación de actualización en `IssueAppService`:

```
public class IssueAppService : ApplicationService, IIssueAppService
{
    private readonly IssueManager _issueManager;
    private readonly IRepository<Issue, Guid> _issueRepository;
    private readonly IRepository<AppUser, Guid> _userRepository;

    public IssueAppService(
        IssueManager issueManager,
        IRepository<Issue, Guid> issueRepository,
        IRepository<AppUser, Guid> userRepository)
    {
        _issueManager = issueManager;
        _issueRepository = issueRepository;
        _userRepository = userRepository;
    }

    public async Task<IssueDto> UpdateAsync(Guid id, UpdateIssueDto input)
    {
        // Get entity from database
        var issue = await _issueRepository.GetAsync(id);

        // Change Title
        await _issueManager.ChangeTitleAsync(issue, input.Title);

        // Change Assigned User
        if (input.AssignedUserId.HasValue)
        {
            var user = await _userRepository.GetAsync(input.AssignedUserId.Value);
            await _issueManager.AssignToAsync(issue, user);
        }

        // Change Text (no business rule, all values accepted)
        issue.Text = input.Text;

        // Update entity in the database
        await _issueRepository.UpdateAsync(issue);

        // Return a DTO represents the new Issue
        return ObjectMapper.Map<Issue, IssueDto>(issue);
    }
}
```

- El método `UpdateAsync` obtiene el ID como un parámetro independiente. No está incluido en `UpdateIssueDto`. Esta es una decisión de diseño que ayuda a ABP a definir correctamente las rutas HTTP cuando se expone automáticamente. Este servicio es un punto final de API HTTP, por lo que no está relacionado con DDD.
- Comienza obteniendo la entidad `Issue` de la base de datos.
- Utiliza `ChangeTitleAsync` de `IssueManager` en lugar de llamar directamente a `IssueSetTitle(...)`. Debido a que necesitamos implementar la verificación de títulos duplicados como se hizo en la creación de entidades, esto requiere algunos cambios en las clases `Issue` y `IssueManager` (se explicarán a continuación).
- Utiliza el método `AssignToAsync` de `IssueManager` si se cambia el usuario asignado con esta solicitud.
- Establece directamente `Issue.Text`, ya que no hay una regla comercial para eso. Si lo necesitamos más adelante, siempre podemos refactorizarlo.
- Guarda los cambios en la base de datos. Nuevamente, guardar las entidades modificadas es responsabilidad del método del Servicio de aplicación que coordina los objetos comerciales y la transacción. Si `IssueManager` hubiera guardado internamente en el método `ChangeTitleAsync` y `AssignToAsync`, habría una operación de base de datos doble (consulte la Discusión: ¿Por qué el problema no se guarda en la base de datos en `IssueManager`? más arriba).

- Finalmente, utiliza `IObjectMapper` para devolver un `IssueDto` que se crea automáticamente mediante el mapeo desde la entidad `Issue` actualizada .

Como dijimos, necesitamos algunos cambios en las clases `Issue` y `IssueManager` .

Primero, haga que `SetTitle` sea interno en la clase `Issue` :

```
internal void SetTitle(string title)
{
    Title = Check.NotNullOrWhiteSpace(title, nameof(title));
}
```

Luego se agregó un nuevo método al `IssueManager` para cambiar el Título:

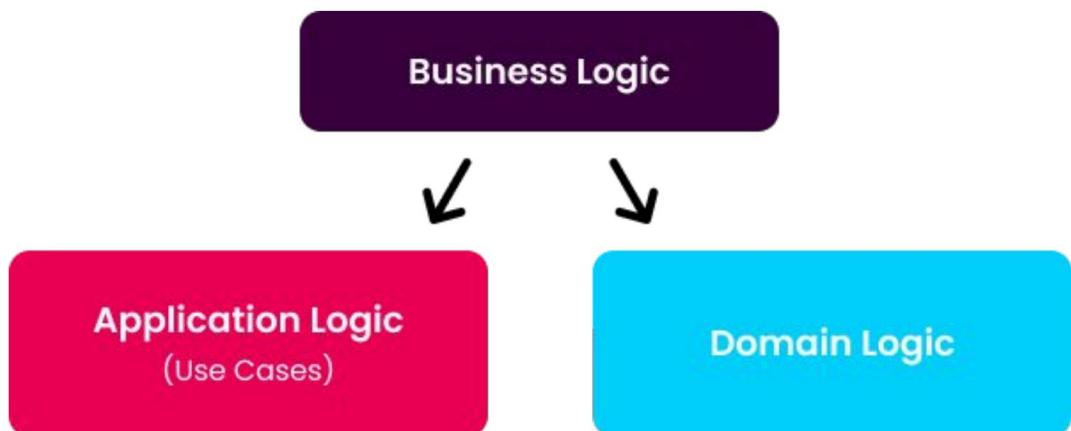
```
public async Task ChangeTitleAsync(Issue issue, string title)
{
    if (issue.Title == title)
    {
        return;
    }

    if (await _issueRepository.AnyAsync(i => i.Title == title))
    {
        throw new BusinessException("IssueTracking:IssueWithSameTitleExists");
    }

    issue.SetTitle(title);
}
```

# Lógica de dominio y lógica de aplicación

Como se mencionó anteriormente, la lógica empresarial en el dominio impulsado. El diseño se divide en dos partes (capas): lógica de dominio y lógica de la aplicación:



La lógica de dominio consta de las reglas de dominio principales del sistema, mientras que la lógica de aplicación implementa casos de uso específicos de la aplicación.

Si bien la definición es clara, la implementación puede no ser fácil. Es posible que no sepas qué código debería estar en la capa de aplicación y cuál en la capa de dominio.

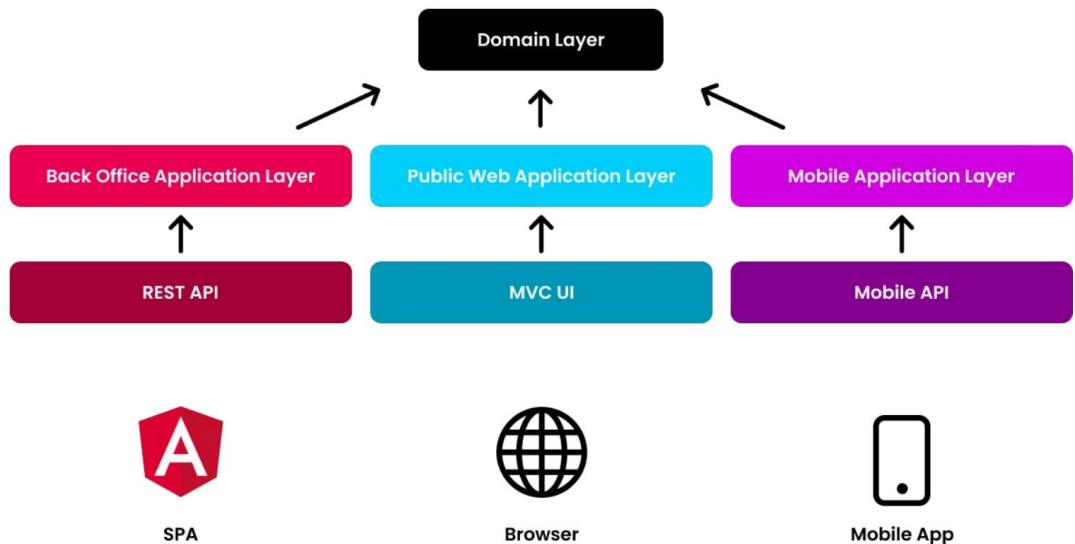
Esta sección intenta explicar las diferencias.

# Múltiples capas de aplicación

DDD ayuda a lidiar con la complejidad cuando su sistema es grande. En especial, si se desarrollan múltiples aplicaciones en un solo dominio, la separación entre lógica de dominio y lógica de aplicación se vuelve mucho más importante.

Supongamos que está construyendo un sistema que tiene múltiples aplicaciones;

- Una aplicación de sitio web público, creada con ASP.NET Core MVC, para mostrar sus productos a los usuarios. Este tipo de sitio web no requiere autenticación para ver los productos. Los usuarios inician sesión en el sitio web solo si realizan alguna acción (como agregar un producto al carrito).
- Una aplicación de back office, creada con Angular UI (que utiliza API REST). Esta aplicación la utilizan los empleados de oficina de la empresa para administrar el sistema (por ejemplo, editar descripciones de productos).
- Una aplicación móvil que tiene una interfaz de usuario mucho más sencilla. En comparación con el sitio web público, puede comunicarse con el servidor a través de API REST u otra tecnología (como sockets TCP).



Cada aplicación tendrá diferentes **requisitos**, diferentes **casos de uso** (métodos de servicio de aplicación), diferentes **DTO**, diferentes **reglas de validación** y **autorización** , etc.

Mezclar todas estas lógicas en una sola capa de aplicación hace que sus servicios contengan demasiadas **condiciones if** con una lógica de negocios complicada que hace que su código sea más difícil de desarrollar, mantener y probar , y conduce a posibles errores.

Si tiene varias aplicaciones con un solo dominio;

- Cree **capas de aplicación independientes** para cada una tipo de aplicación/cliente e implementar la lógica de negocios específica de la aplicación en estas capas separadas.
- Utilice una **única capa de dominio** para compartir la lógica del dominio principal.

Este diseño hace que sea aún más importante distinguir entre lógica de dominio y lógica de aplicación.

Para que la implementación sea más clara, puedes crear diferentes proyectos (`.csproj`) para cada tipo de aplicación. Por ejemplo:

- Proyectos `IssueTracker.Admin.Application` y `IssueTracker.Admin.Application.Contracts` para la aplicación Back Office (administración).
- Proyectos `IssueTracker.Public.Application` y `IssueTracker.Public.Application.Contracts` para la aplicación web pública.
- Proyectos `IssueTracker.Mobile.Application` y `IssueTracker.Mobile.Application.Contracts` para la aplicación móvil.

## Ejemplos

Esta sección contiene algunos ejemplos de servicios de aplicación y servicios de dominio para analizar cómo decidir colocar la lógica empresarial dentro de estos servicios.

## Ejemplo: Creación de una nueva organización en un servicio de dominio

```
public class OrganizationManager : DomainService
{
    private readonly IRepository<Organization> _organizationRepository;
    private readonly ICurrentUser _currentUser;
    private readonly IAuthorizationService _authorizationService;
    private readonly IEmailSender _emailSender;

    public OrganizationManager(
        IRepository<Organization> organizationRepository,
        ICurrentUser currentUser,
        IAuthorizationService authorizationService,
        IEmailSender emailSender)
    {
        _organizationRepository = organizationRepository;
        _currentUser = currentUser;
        _authorizationService = authorizationService;
        _emailSender = emailSender;
    }

    public async Task<Organization> CreateAsync(string name)
    {
        if (await _organizationRepository.AnyAsync(x => x.Name == name))
        {
            throw new BusinessException("IssueTracking:DuplicateOrganizationName");
        }

        await _authorizationService.CheckAsync("OrganizationCreationPermission");

        Logger.LogDebug($"Creating organization {name} by {_currentUser.UserName}");

        var organization = new Organization();

        await _emailSender.SendAsync(
            "systemadmin@issuetracking.com",
            "New Organization",
            "A new organization created with name: " + name
        );

        return organization;
    }
}
```

Veamos el **método CreateAsync** paso a paso para discutir si la parte del código debe estar en el Servicio de Dominio o no;

- **CORRECTO:** Primero verifica si hay **nombres de organizaciones duplicados** y, en este caso, genera una excepción. Esto está relacionado con la regla del dominio principal y nunca permitimos nombres duplicados.
  
- **INCORRECTO:** Los servicios de dominio no deberían funcionar Autorización. Autorización debe hacerse en el Capa de aplicación.
  
- **INCORRECTO:** Registra un mensaje que incluye el **valor actual** Del usuario Nombre de usuario. El servicio de dominio no debe depender del usuario actual. Los servicios de dominio deben poder utilizarse. Incluso si no hay ningún usuario en el sistema, el usuario actual (sesión) debe ser un concepto relacionado con la capa de presentación/aplicación.
  
- **INCORRECTO:** Envía un correo electrónico Acerca de esta nueva organización Creación. Creemos que esta también es una lógica empresarial específica de cada caso de uso. Es posible que desee crear diferentes tipos de correos electrónicos en diferentes casos de uso o que no necesite enviar correos electrónicos en algunos casos.

## Ejemplo: Creación de una nueva organización en una aplicación Servicio

```
public class OrganizationAppService : ApplicationService
{
    private readonly OrganizationManager _organizationManager;
    private readonly IPaymentService _paymentService;
    private readonly IEmailSender _emailSender;

    public OrganizationAppService(
        OrganizationManager organizationManager,
        IPaymentService paymentService,
        IEmailSender emailSender)
    {
        _organizationManager = organizationManager;
        _paymentService = paymentService;
        _emailSender = emailSender;
    }

    [UnitOfWork]
    [Authorize("OrganizationCreationPermission")]
    public async Task<Organization> CreateAsync(CreateOrganizationDto input)
    {
        await _paymentService.ChargeAsync(
            CurrentUser.Id,
            GetOrganizationPrice()
        );

        var organization = await _organizationManager.CreateAsync(input.Name);

        await _organizationManager.InsertAsync(organization);

        await _emailSender.SendAsync(
            "systemadmin@issuetracking.com",
            "New Organization",
            "A new organization created with name: " + input.Name
        );
    }

    private double GetOrganizationPrice()
    {
        return 42; //Gets from somewhere else...
    }
}
```

Veamos el **método CreateAsync** paso a paso para discutir si la parte del código debe estar en el Servicio de Aplicación o no;

- **CORRECTO:** Los métodos de servicio de aplicación deben ser una unidad de trabajo (transaccional). Unidad de trabajo de ABP El sistema hace esto automáticamente (incluso sin necesidad de agregar el atributo **[UnitOfWork]** para los Servicios de Aplicación).
- **CORRECTO:** Autorización debe hacerse en el Capa de aplicación. Aquí, esto se hace mediante el atributo **[Autorizar]**.
- **CORRECTO:** Se denomina pago (un servicio de infraestructura) al cobro de dinero por esta operación (Crear una Organización es un servicio pago en nuestro negocio).
- **CORRECTO:** El método de servicio de aplicación es responsable de guardar cambios en la base de datos.
- **CORRECTO:** Podemos enviar correo electrónico como notificación a la administrador del sistema.
- **INCORRECTO:** No devolver entidades de la aplicación Servicios. Devuelva un DTO en su lugar.

## Discusión: ¿Por qué no trasladamos la lógica de pago al servicio de dominio?

Quizás te preguntes por qué el código de pago no está dentro de

**OrganizationManager.** Es algo importante y nunca queremos perder el pago.

Sin embargo, el hecho de que sea importante no es suficiente para considerar un código como lógica empresarial básica. Es posible que tengamos otros casos de uso en los que no cobremos dinero por crear una nueva organización.

Ejemplos:

- Un usuario administrador puede utilizar una aplicación de Back Office para crear una nueva organización sin ningún pago.
- Un sistema de importación/integración/sincronización de datos que funcione en segundo plano también puede necesitar crear organizaciones sin ninguna operación de pago.

Como puede ver, el pago no es una operación necesaria para crear una organización válida. Es una lógica de aplicación específica del caso de uso.

## Ejemplo: Operaciones CRUD

```
public class IssueAppService
{
    private readonly IssueManager _issueManager;

    public IssueAppService(IssueManager issueManager)
    {
        _issueManager = issueManager;
    }

    public async Task<IssueDto> GetAsync(Guid id)
    {
        return await _issueManager.GetAsync(id);
    }

    public async Task CreateAsync(IssueCreationDto input)
    {
        await _issueManager.CreateAsync(input);
    }

    public async Task UpdateAsync(UpdateIssueDto input)
    {
        await _issueManager.UpdateAsync(input);
    }

    public async Task DeleteAsync(Guid id)
    {
        await _issueManager.DeleteAsync(id);
    }
}
```

Este servicio de aplicación no hace nada por sí mismo y delega todo el trabajo al servicio de dominio. Incluso pasa los DTO al **IssueManager**.

- No cree métodos de servicio de dominio solo por simplicidad.

Operaciones CRUD sin ninguna lógica de dominio.

- Nunca pase DTO a ni devuelva DTO del dominio

Servicios.

Los servicios de aplicación pueden trabajar directamente con repositorios para consultar, crear, actualizar o eliminar datos a menos que existan algunos. Durante estas operaciones se deben realizar las lógicas de dominio.

En tales casos, cree métodos de Servicio de Dominio, pero solo para aquellos Realmente necesario.

¡No cree dichos métodos de servicio de dominio CRUD simplemente pensando que pueden ser necesarios en el futuro ([YAGNI!](#))! Hazlo cuando lo necesites y refactoriza el código existente. Dado que la capa de aplicación abstrae elegantemente la capa de dominio, el proceso de refactorización no afecta a la capa de interfaz de usuario ni a otros cli

## Libros de referencia

Si está más interesado en el diseño impulsado por el dominio y la construcción de sistemas empresariales a gran escala, se recomiendan los siguientes libros como libros de referencia;

- "Diseño impulsado por el dominio" de Eric Evans
- "Implementación de diseño impulsado por dominio" por Vaughn Vernon
- "Arquitectura limpia" de Robert C. Martin

The Official Guide

# Mastering ABP Framework

Building maintainable .NET solutions by following software development best practices using ABP



By  
Halil Ibrahim KALKAN

You can buy from

**amazon**



A complete  
**WEB DEVELOPMENT  
PLATFORM**

built-on abp framework

ABP Commercial provides pre-built application modules, rapid application development tooling, professional UI themes, premium support and more.

VISIT WEBSITE →



APPLICATION  
MODULES

DEVELOPER  
TOOLS



UI  
THEMES



STARTUP  
TEMPLATES



PREMIUM  
SUPPORT



ADDITIONAL  
SERVICES



[www.abp.io](http://www.abp.io)