

Clean Architecture in .NET Core: Step by Step

Olimpo Bonilla Ramírez

TABLA DE CONTENIDO

INTRODUCCIÓN	6
Material Requerido.....	7
1. ANTES DE COMENZAR	8
1.1. Problemática.....	8
1.2. Migración fluida.	8
1.3. EntityFramework Fluent Migrator y sus inconvenientes.	8
1.4. Uso de DbUp para Fluent Migration DataBase.	9
1.5. Tips para migración de Base de Datos.	14
1.6. Creando nuestra Base de Datos.	16
2. CLEAN ARCHITECTURE.....	22
2.1. Principios SOLID.	22
2.2. Inyección de Dependencias: conceptos y definiciones.	23
2.3. Inversión de Control (IoC).	25
2.4. Clean Architecture.....	27
2.5. Onion Architecture.....	30
2.6. Hexagonal Architecture.....	31
2.7. Organizando nuestro proyecto.	31
2.8. Creando el esqueleto de nuestra aplicación de Clean Architecture.	33
3. ENTITY FRAMEWORK SCAFFOLD E INYECCIÓN DE DEPENDENCIAS.	37
3.1. ¿Qué es Scaffold?.....	37
3.2. Especificaciones y nomenclatura para la Inyección de Dependencias.....	37
3.3. Scaffolding en .NET Core.....	37
3.4. Ajustes adicionales posterior a Scaffold.....	38
3.5. Creación de Controllers.....	40

3.6. Inyectando dependencias.....	40
3.7. Contenedor de Inversión de Control.....	40
4. INTERFAZ FLUIDA EN ENTITY FRAMEWORK.....	42
4.1. Interfaz Fluida (Fluent Interface).....	42
4.2. ¿Qué es Fluent API de Entity Framework?.....	42
4.3. Ajustando el idioma inglés.....	43
4.4. Configuración de entidades.....	44
4.5. Quitando entidades innecesarias.....	47
4.6. Resumen.....	47
5. CONCEPTOS DE HTTP Y REST.....	48
5.1. Concepto de API.....	48
5.2. REST y API REST.....	48
5.3. RESTFul.....	50
5.4. CRUD.....	50
5.5. HTTP Verbs.....	51
5.6. Códigos de estatus HTTP.....	53
5.7. StateLess.....	54
5.8. Consejos de diseño correcto de API REST.....	54
5.9. Modelo de madurez de Richardson.....	55
5.10. Regresando.....	58
5.11. Resumen.....	60
6. AUTOMAPPER Y DTO.....	61
6.1. Concepto de DTO.....	61
6.2. Concepto de Mapping (Mapeo).....	61
6.3. Técnicas comunes de Mapeo de Datos.....	62
6.4. Casos de uso de Mapeo de Datos.....	63

6.5. AutoMapper.....	63
6.6. Nota importante.	64
6.7. Ajustando a nuestro código fuente.....	64
6.8. Resumen.....	67
7. AUTOFILTERS Y VALIDACIÓN FLUIDA.....	68
7.1. AutoFilters en ASP.NET Core.	68
7.2. Implementar un ActionFilter.....	69
7.3. Ambito de los AutoFilters en .NET Core.....	69
7.4. Orden de invocación en ASP.NET Core.	70
7.5. Fluent Validation.....	71
7.6. Ajustando nuestro código fuente: extendiendo el arranque del proyecto.....	71
7.7. Creando nuestro ActionFilter global.	74
7.8. Usando FluentAPIValidation.	76
7.9. Resumen.....	79
8. PERSISTENCIA DE DATOS: UNIT OF WORK Y REPOSITORY	80
8.1. Persistencia de Datos.	80
8.2. Unit Of Work.	80
8.3. Patrones requeridos a usar.....	81
8.4. Capas necesarias.	81
8.5. Sin embargo...	82
8.6. DRY Principle.	83
8.7. El componente EntityBase.	84
8.8. El principio de Fail-Fast.	86
8.9. Guard Clause Pattern.	86
8.10. Antes de empezar...	89
8.11. Diseñando nuestro núcleo de Persistencia de Datos.....	94

8.12. Empezando por la capa de Dominio.....	94
8.13. Empezando por la capa de Infraestructura.....	101
8.14. Terminando la capa de Presentación.....	108
8.15. Resumen.....	110
9. PERSISTENCIA DE DATOS (II): CQRS, MEDIATOR Y CONTROL DE EXCEPCIONES.....	111
9.1. Mediator Pattern.....	111
9.2. CQRS Pattern.....	112
9.3. MediatR: CQRS y Mediator en un solo lugar.....	113
9.4. Uso básico de MediatR.....	114
9.5. Pipeline en ASP.NET Core.....	116
9.6. Middleware en ASP.NET Core.....	116
9.7. MediatR y Fluentvalidation: reemplazando la validación en ASP.NET Core.	118
9.8. Manejo de excepciones de validación.	119
9.9. Cambios a nuestro proyecto.	120
9.10. Aplicando a nuestro código.....	121
9.11. Conclusión.....	134
APENDICE.....	135
Bibliografía.....	135
APENDICE.....	135
Bibliografía.....	135
APENDICE.....	135
Bibliografía.....	135

INTRODUCCIÓN.

Este documento electrónico tiene como objetivo, explicar paso a paso como desarrollar una aplicación en .NET Core con el modelo de *Clean Architecture* (Arquitectura Limpia), la cual, en estos días, al momento de escribir este documento, lo piden como requisito para los Desarrolladores .NET Full Stack en las empresas para el diseño, implementación y publicación de software, a nivel empresarial y comercial.

A través de los capítulos, estaré explicando de una manera sencilla, en la medida de lo posible, estos conceptos. En Internet, siempre hay ejemplos y muchos de implementaciones de este tipo de proyectos para aplicaciones backend como Web API, API Rest y RESTFul. Sin embargo, quienes desarrollan estas implementaciones explican poco o muy vagamente el por que diseñan el código fuente a su manera, sin justificar claramente si aplicaron los principios de esta metodología de arquitectura de software y está dirigido para aquellos que no tienen experiencia previa en este modo de desarrollo, pero también, por qué no, sirve para aquellos que ya conocen grosso modo esta forma de hacer código fuente. Muchos de los conceptos que piden en las empresas, cuando hacen las entrevistas técnicas de parte de Recursos Humanos y el área de Tecnologías de Información, a veces no los conocemos pero este documento concentrará, en la mejor medida posible, el significado de los mismos.

Esperemos que este documento ayude a los desarrolladores de software tener los conceptos claros de buenas prácticas de programación que les ayudará a ser mejores desarrolladores cada día y aplicarlos de manera eficiente para los problemas de la vida real.

Material Requerido.

En Windows, usaremos los siguientes programas para este curso:

- Microsoft Visual Studio 2019 (en adelante).
- Microsoft SQL Server 2019 (en adelante) o cualquier gestor de Base de Datos.
- Control de versiones Git y su aplicación cliente (Git Extensions, SourceTree, etc).
- Docker (Opcional).
- Postman para validar API Rest. Tambien se puede usar Insomnia, RESTer, Test Studio de Telerik, etc.

Si está usando Linux o MAC:

- Visual Studio Code.
- Microsoft .NET Core SDK 3.1 en adelante.
- Cualquier gestor de Base de Datos (MySQL o MariaDB, PostgreSQL, etc).
- Control de versiones Git y su aplicación cliente (Git Ahead, SourceTree, etc).
- Docker (Opcional).
- Postman para validar API Rest.

Independientemente de que versiones de los programas sean, el código fuente y la forma de escribirlo es la misma, a menos de que la evolución de la sintaxis de C# cambie en el transcurso de los años venideros.

1. Antes de comenzar.

Para empezar a trabajar con el tema de Clean Architecture, comenzaremos a crear primero nuestra herramienta de migración de Base de Datos SQL, que es importante en una aplicación Backend en .NET Core.

1.1. Problemática.

Para los efectos de este tutorial, vamos a plantearnos un problema sencillo en la cual, podamos usar los conceptos que vamos a ver más adelante sobre programación.

Nuestro problema es:

Una empresa de abarrotes en México llamada PATOSA S.A. de C.V. acaba de ser creada y su giro es el área comercial. Los socios y ejecutivos tienen pensado abrir sus primeras tres sucursales para vender sus productos, los cuales, tiene almacenado en su Centro de Distribución (CEDI). La empresa actualmente tiene 3 sucursales: una en la Ciudad de México, otra en Monterrey y otra en Guadalajara. El caso es que el CEDI, quiere tener el control de su inventario para distribuir los artículos a las tres sucursales antes mencionadas y que cada artículo tiene su precio de venta en la sucursal donde va a venderse.

Se necesita lo siguiente:

- Tener un catálogo de sucursales.
- Tener un catálogo de artículos, el cual debe tener los siguientes datos: el SKU (o identificador del artículo), nombre del artículo, descripción del mismo, precio unitario, identificador del tipo de artículo, y el identificador de la sucursal a donde se va a mandar para su venta, desde el CEDI.
- Tener un catálogo de tipos de artículo.
- Tener un catálogo de cuentas de usuario para llevar el control de la captura de datos, aplicado para todos los catálogos antes mencionados.

Grosso modo, este sería un pequeño, pero interesante ejercicio que haremos a lo largo de este documento.

1.2. Migración fluida.

Mientras desarrollamos una aplicación, administraremos la base de datos manualmente, es decir, hacemos scripts SQL (para crear y actualizar tablas, SPs, funciones, etc.) y luego los ejecutamos y también necesitamos administrarlos en un orden determinado para que se pueda ejecutar. en el entorno superior sin problemas. Por lo tanto, administrar estos cambios en la base de datos con un desarrollo y una implementación regulares es una tarea difícil. Y la cuestión de mantenibilidad, ni se diga.

Ahora, la buena noticia es que existen diversos componentes para .NET Core para resolver todos los problemas mencionados y que a través de herramientas como Git, podemos darme un seguimiento adecuado y eficiente. Estas herramientas son: EntityFramework Core, Fluent Migrator, DbUp, entre otros.

1.3. EntityFramework Fluent Migrator y sus inconvenientes.

EntityFramework es el componente de migración fluida de Base de Datos por excelencia de Microsoft. A pesar de que muchos desarrolladores lo usan para realizar procesos de migración fluida para Bases de Datos, resulta que tiene algunos inconvenientes:

- Crecimiento de la Base de Datos de manera considerable, puesto que el log de migraciones crece y se guarda una replica en tipo de dato BLOB, de la migración aplicada en Base de Datos. Si la Base de Datos es concurrente, puede generarse problemas de rendimiento en ambientes de Producción o QA.
- Se necesita mucho código en C# para generar las migraciones. Pocas veces se usa el lenguaje SQL para hacerlas. En Code First, es complicado darle un seguimiento, por que EF asigna en automático los tipos de datos para las entidades, lo cual, a veces es conflictivo cuando se trata de procesar las consultas de manera rápida.
- Si una migración se aplica y falla, se tiene que ejecutar comandos para revertirlas, lo cual, puede afectar la integridad de la Base de Datos y a veces engorroso, puesto que esas migraciones dependen de otras migraciones aplicadas.

En mi opinión personal, no usaría ese procedimiento con Entity Framework Core para crear y mantener la Base de Datos de una aplicación backend. Afortunadamente existen otras alternativas mas sencillas que Entity Framework Core para *Fluent Migration DataBase*.

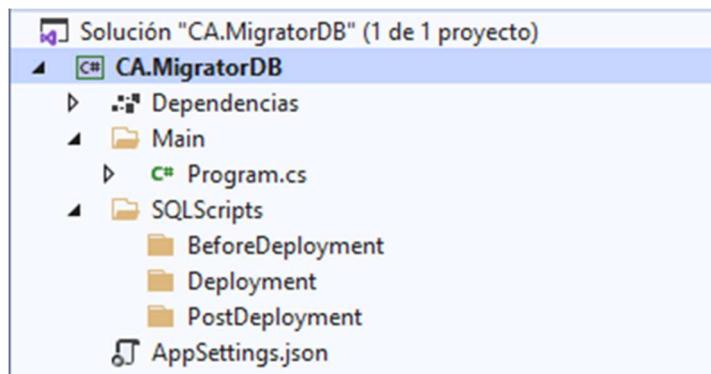
1.4. Uso de DbUp para Fluent Migration DataBase.

DbUp es una biblioteca .NET Core de código abierto que nos proporciona una forma de implementar cambios en la base de datos. Además, rastrea qué scripts SQL ya se han ejecutado, tiene muchos proveedores de scripts SQL disponibles y otras características interesantes como el preprocesamiento de scripts.

Y la pregunta del millón es:

¿Por qué *DbUp* y no *EntityFramework Migrations* o *Fluent Migrator*?

Bueno, si no queremos que C# genere automáticamente SQL, sin mover nada y usar nuestros conocimientos de SQL, o bien, no tenemos conocimientos firmes en Code First o POCO, este componente es el ideal por que es una solución mas pura con un lenguaje sencillo y creo que no necesitamos una herramienta especial para generarlo. Para hacer esto, abramos Visual Studio y creamos una aplicación de consola en .NET Core 3.1 en adelante llamado **CA.MigratorDB** y hagamos la estructura del proyecto como se muestra a continuación:



Nuestra estructura es la siguiente:

- El archivo de configuración **AppSettings.json**, el cual, tendrá la cadena de conexión a Base de Datos.
- La carpeta **Main** es donde se tiene el arranque de la aplicación.
- La carpeta **Scripts** es donde se tiene que guardar los archivos SQL necesarios para nuestra migración. En ese orden, tenemos las siguientes tres subcarpetas:
 - **BeforeDeployment**. Operaciones antes de deployment definitivo en Base de Datos. Aquí se pueden crear cuentas de usuario, esquemas o inicios de sesión relacionados a la base de datos, permisos sobre objetos, etc. Solo se ejecutan una sola vez.

- **Deployment.** Operaciones para crear objetos de Base de Datos y carga de datos de las mismas, en especial, tablas y store procedures. Solo se ejecutan una sola vez.
- **PostDeployment.** Operaciones para probar los objetos de Base de Datos ya hechos. Se ejecutan multiples veces para probar y verificar su correcto funcionamiento.

Guardemos los cambios y ahora ejecutemos desde la *Consola de Administración de Paquetes de Visual Studio*, los siguientes comandos, en el siguiente orden:

```
$ dotnet add package dbup-core
$ dotnet add package dbup-sqlserver
$ dotnet add package dbup-mysql
$ dotnet add package Microsoft.Extensions.Configuration
$ dotnet add package Microsoft.Extensions.Configuration.Binder
$ dotnet add package Microsoft.Extensions.Configuration.Abstractions
$ dotnet add package Microsoft.Extensions.Configuration.FileExtensions
$ dotnet add package Microsoft.Extensions.Configuration.JSON
$ dotnet add package Microsoft.Extensions.DependencyInjection
$ dotnet add package Microsoft.Extensions.DependencyInjection.Abstractions
$ dotnet add package Microsoft.Extensions.Options.ConfigurationExtensions
```

Aquí estamos integrando los componentes de .NET Core para la inyección de dependencias y el uso de DbUp para SQL Server y MySQL. En el archivo de proyecto CA.Migrator.csproj, agregue las siguientes líneas para que cuando se publique la aplicación de consola para contenedores Docker, se migre la configuración de la cadena de conexión a la Base de Datos, según el ambiente de desarrollo que se aplique:

```
<!-- Carpetas y archivos adicionales que se deben de publicar cuando es modo Debug o Release. -->
<ItemGroup>
  <None Update="AppSettings.json" CopyToOutputDirectory="Always" CopyToPublishDirectory="Always" />
</ItemGroup>
```

Guardemos los cambios y ahora, creamos los siguientes archivos en la carpeta Main, para configurar DbUp: creamos primero un archivo de clase llamado ConnectionStringCollection.cs el cual, guarda las cadenas de conexión a Base de Datos:

```
using System;
namespace CA.MigratorDB
{
  [Serializable]
  public class ConnectionStringCollection
  {
    public string ConnectionStringSQLServer { get; set; }
    public string ConnectionStringMySQLServer { get; set; }
  }
}
```

Después, el archivo **Program.cs** debe tener algo como esto:

```
using System.IO;
using System.Threading.Tasks;

using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace CA.MigratorDB
{
  class Program
  {
    static async Task Main(string[] args)
    {
      var services = ConfigureServices();
      var serviceProvider = services.BuildServiceProvider();
      await serviceProvider.GetService<App>().RunAsync(args);
    }

    public static IConfiguration LCAdConfiguration()
    {
      var builder = new ConfigurationBuilder().SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("AppSettings.json", optional: true,
                    relCAdOnChange: true);
      return builder.Build();
    }

    private static IServiceCollection ConfigureServices()
    {
      IServiceCollection services = new ServiceCollection();
```

```

    var config = LCAdConfiguration();
    services.AddSingleton(config);

    /* Lectura de opciones del archivo de configuración.*/
    services.Configure<ConnectionStringCollection>(options =>
        config.GetSection($"Collection.ConnectionStrings").Bind(options));

    /* Inyectamos la clase 'App' */
    services.AddSingleton<App>();

    /* Otros servicios de la aplicación de la consola.*/
    /* services.AddTransient<IUser, User>(); */

    return services;
}
}

```

Este archivo solo emulamos la inyección de dependencias como si fuera una aplicación en ASP.NET Core al arranque del mismo. Es sencillo: estamos realizando la inyección de dependencias de las abstracciones para las implementaciones.

Al final, el archivo **App.cs** debe tener algo como esto:

```

using System;
using System.Threading;
using System.Reflection;
using System.Threading.Tasks;

using Microsoft.Extensions.Options;
using Microsoft.Extensions.Configuration;

using DbUp;
using DbUp.Engine;
using DbUp.Support;

namespace CA.MigratorDB
{
    public class App
    {
        private readonly ConnectionStringCollection _settings;

        public App(IOptions<ConnectionStringCollection> settings) { _settings = settings.Value; }

        public async Task RunAsync(string[] args)
        {
            try
            {
                /* Inicio de la tarea asíncrona.*/
                await Task.Run(() => {
                    /* Cadena de conexión a la Base de Datos tomada desde el archivo AppConfig.json.*/
                    var connectionString = _settings.ConnectionStringSQLServer;

                    /* Creamos la Base de Datos, si no existe...*/
                    EnsureDatabase.For.SqlDatabase(connectionString);

                    /* Configuramos el motor de migración de Base de Datos de DbUp.*/
                    var upgradeEngineBuilder = DeployChanges.To.SqlDatabase(connectionString, null)
                        // Pre-deployment scripts: configurarlos para que siempre se ejecuten en el primer grupo.
                        .WithScriptsEmbeddedInAssembly(Assembly.GetExecutingAssembly(), x =>
                            x.StartsWith($"CA.MigratorDB.SQLScripts.BeforeDeployment."),
                            new SqlScriptOptions { ScriptType = ScriptType.RunAlways, RunGroupOrder = 0 });
                    // Main Deployment scripts: se ejecutan solo una vez y corren en el segundo grupo.
                    .WithScriptsEmbeddedInAssembly(Assembly.GetExecutingAssembly(), x =>
                            x.StartsWith($"CA.MigratorDB.SQLScripts.Deployment."),
                            new SqlScriptOptions { ScriptType = ScriptType.RunOnce, RunGroupOrder = 1 });
                    // Post deployment scripts: siempre se ejecutan estos scripts después de que todo se haya implementado.
                    .WithScriptsEmbeddedInAssembly(Assembly.GetExecutingAssembly(), x =>
                            x.StartsWith($"CA.MigratorDB.SQLScripts.PostDeployment."),
                            new SqlScriptOptions { ScriptType = ScriptType.RunAlways, RunGroupOrder = 2 });
                    // De forma predeterminada, todos los scripts se ejecutan en la misma transacción.
                    .WithTransactionPerScript()
                    // Colocar el log en la consola.
                    .LogToConsole();

                    /* Construimos el proceso de migración.*/
                    Console.WriteLine($"Aplicando cambios en Base de Datos...");
                    var upgrader = upgradeEngineBuilder.Build();

                    if (upgrader.IsUpgradeRequired())
                    {
                        var result = upgrader.PerformUpgrade();

                        /* Mostrar el resultado.*/
                        if (result.Successful)
                        {

```

```
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine($"Ejecución satisfactoria de la migración a Base de Datos.");
    }
    else
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine($"La migración de Base de Datos falló. Revise el siguiente mensaje de error.");
        Console.WriteLine(result.Error);
    }

    Console.ResetColor();
}

Thread.Sleep(500);
}).ConfigureAwait(false);
}
catch (Exception oEx)
{
    Console.WriteLine($"Ocurrió un error al realizar este proceso de migración de Base de Datos: {oEx.Message.Trim()}.");
}
finally
{
    Console.WriteLine($"Pulse cualquier tecla para salir..."); Console.ReadLine();
}
}
}
```

Expliquemos las siguientes líneas:

```
EnsureDatabase.For.SqlDatabase(connectionString);
```

indica que si la Base de Datos no existe, se crea en el servidor de Base de Datos.

```
DeployChanges.To.SqlDatabase(connectionString, null)
```

Esta función indica que se realizará el deployment para la ejecución de los scripts SQL contenidos en el proyecto, en la carpeta **SQLScripts**. La opción **SqlDataBase** es para apuntar a un servidor de SQL Server. Si fuera MySQL o MariaDB, sería el método **MySQLDatabase**. En cualquiera de ambos casos, se aplica la migración.

```
WithScriptsEmbeddedInAssembly(Assembly.GetExecutingAssembly(), x =>
    x.StartsWith($"CA.MigratorDB.SQLScripts.BeforeDeployment."),
    new SqlScriptOptions { ScriptType = ScriptType.RunOnce, RunGroupOrder = 0 })
```

Esto indica que se van a ejecutar los scripts SQL que existen en la carpeta **BeforeDeployment** del ensamblado del proyecto. El tipo de ejecución será una sola vez, puesto que el valor **ScriptType** define si el script se ejecuta una vez (**RunOnce**) o varias veces (**RunAlways**). El valor **RunGroupOrder** es necesario definirlo para indicar el orden de ejecución de los scripts en el proceso de migración. Se empieza desde 0.

Las siguientes líneas

```
// Main Deployment scripts: se ejecutan solo una vez y corren en el segundo grupo.  
.WithScriptsEmbeddedInAssembly(Assembly.GetExecutingAssembly(), x =>  
    x.StartsWith("CA.MigratorDB.SQLScripts.Deployment."),  
    new SqlScriptOptions { ScriptType = ScriptType.RunOnce, RunGroupOrder = 1 })  
  
// Post deployment scripts: siempre se ejecutan estos scripts después de que todo se haya implementado.  
.WithScriptsEmbeddedInAssembly(Assembly.GetExecutingAssembly(), x =>  
    x.StartsWith("CA.MigratorDB.SQLScripts.PostDeployment."),  
    new SqlScriptOptions { ScriptType = ScriptType.RunAlways, RunGroupOrder = 2 })
```

hacen lo mismo, tomando los scripts de las carpetas **Deployment** y **PostDeployment**, con su número de orden establecido. Las siguientes líneas:

```
// De forma predeterminada, todos los scripts se ejecutan en la misma transacción.  
.WithTransactionPerScript()  
// Colocar el log en la consola.  
.LogToConsole();
```

Indican que toda la ejecución de los scripts se hará por transacciones y se mostrará el resultado en consola.

```
var upgrader = upgradeEngineBuilder.Build();
```

Indica la variable `upgrader` es un objeto `UpgradeEngine`, el cual, gestiona y construye el proceso de ejecución de los scripts.

```
if (upgrader.IsUpgradeRequired())
{
    var result = upgrader.PerformUpgrade();

    /* Mostrar el resultado.*/
    if (result.Successful)
    {
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine($"Ejecución satisfactoria de la migración a Base de Datos.");
    }
    else
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine($"La migración de Base de Datos falló. Revise el siguiente mensaje de error.");
        Console.WriteLine(result.Error);
    }

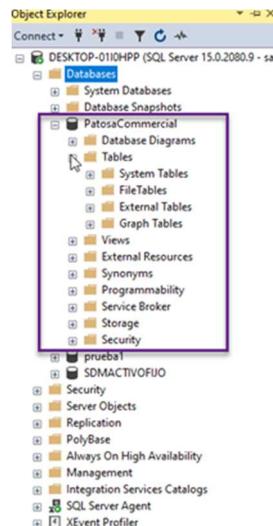
    Console.ResetColor();
}
```

Este bloque finalmente hace la conclusión del proceso de migración de Base de Datos. La función `PerformUpgrade` ejecuta todo el lote de scripts almacenados en el objeto `upgrader` y genera un resultado. Si es `Successful`, quiere decir que se aplicaron correctamente los cambios en la Base de Datos, de lo contrario, lanza una excepción indicando el error generado durante el proceso.

Configuremos el archivo `AppSettings.json` y debe quedar algo como esto:

```
{
    "ConnectionStrings": {
        "DefaultConnectionString": "Server=localhost;Database=PatosaCommercial;User Id=sa;Password=mypassword;"
    },
    "CollectionConnectionStrings": {
        "ConnectionStringSqlServer": "Server=localhost;Database=PatosaCommercial;Integrated Security=True;",
        "ConnectionStringMySQLServer": "Server=localhost;Database=PatosaCommercial;User Id=sa;Password=mypassword;"
    }
}
```

Guardemos los cambios y comilemos. Al ejecutarse nuestra aplicación de consola, vemos que no se aplicaron cambios pero si revisamos en el servidor de Base de Datos, notamos que se creó una base de datos nueva con el nombre de **PatosaComercial**, el cual, muestra que DbUp aplicó los cambios de manera satisfactoria.



En este caso, fue en un servidor de SQL Server. Si maneja otro gestor de Base de Datos, tiene que verificar que en efecto se creó esa Base de Datos. Hasta el momento, la Base de Datos está vacía, pero nuestra aplicación está lista para que integremos los scripts SQL que queramos.

1.5. Tips para migración de Base de Datos.

Estos tips son importantes, si queremos ejecutar migraciones usando DbUp, FluentMigrator, Ef Migrations o cualquier otra herramienta es realmente fácil de comenzar. Con algunos consejos, puede sobrevivir con éxito a un proyecto de larga duración sin estrés ni experiencias tristes.

1. **Asegúrese de no perder los datos.** Imagine que almacena la contraseña de usuario en la base de datos como una cadena simple (solo imagine que no lo haga). Ahora ha llegado el momento de arreglar esta, digamos, extraña situación. Entonces, desea hacer un hash md5 en las contraseñas. Preparamos el script de migración, actualizamos el código base y se ejecuta todo. Boom: algo se bloqueó en la aplicación, por lo que debe restaurar rápidamente la versión anterior para que la empresa aún pueda ganar dinero con sus clientes. Desafortunadamente las contraseñas tienen hash y no hay vuelta atrás. Acaba de causar un retraso mayor de lo que debería (probablemente tenga que restaurar una copia de seguridad y perder datos nuevos o corregir el error en la aplicación y hacer que todos esperen). En tales casos, por favor:

- Prepare un script de migración que amplíe la tabla con una columna más y mantenga la antigua.
- Actualizar el código de la aplicación.
- Implementar la aplicación.
- Si todo funciona, cree nuevas migraciones que eliminan la columna anterior.

Esto puede significar que no le rechazarán el próximo aumento de sueldo.

2. **No modificar o eliminar sus scripts.** Todos, en algún momento, lo hemos hecho como desarrolladores novatos. A veces, modificamos el script o los eliminamos después para crear nuevos con el fin de que se vean bonitos y funcionales. Aquí, por cuestión de salud mental, no haga semejante cosa. Aunque sean muchos archivos de scripts SQL en un proceso de migración, **se tienen que conservar para futuras referencias**. Todo lo que tiene que hacer es introducir una nueva transición del estado A al estado B. Si la cantidad de archivos es molesta, simplemente colóquelos en un directorio (por ejemplo, con el nombre del año: 2020, 2021, etc.).
3. **Utilice marcas de tiempo en formato UNIX para el control de versiones.** Recuerde que todo lo que desarrollamos y compilamos se guarda en un repositorio de Git (no voy a entrar a detalles de que es Git y control de versiones) y que debemos tener *un orden adecuado en nuestros proyectos*, dicho de otra forma, tengamos nuestra carpeta de archivos SQL en orden y de manera limpia.

Un patrón de nomenclatura común que usamos para la nomenclatura de migración es un *entero secuencial + descripción*. Por ejemplo:

```
1-AddCustomersTable.sql  
2-AddOrdersTable.sql
```

Esto funciona bien cuando es un proyecto pequeño, pero cuando se trata de proyectos grandes, tarde o temprano terminaría así:

```
1-AddCustomersTable.sql  
2-AddOrdersTable.sql  
2-AddSuppliersTable.sql  
4-ExtendCustomersTableWithName.sql
```

Esto es una mala práctica de programación y teniéndolo así, hace difícil su rastreabilidad, a la hora de buscar referencia histórica de un objeto de Base de Datos para resolver conflictos que puedan ocurrir. La mejor alternativa es usar marcas de tiempo en formato UNIX, como la siguiente:

```
1607176125-AddCustomerTable.sql  
1607912575-AddOrdersTable.sql
```

1607976875-AddSuppliersTable.sql

En este [enlace](#) podemos usar la conversión de la fecha actual a marca de tiempo UNIX.

4. **No realice cambios en la base de datos fuera de su migrador.** Hay algunas herramientas interesantes en Azure, como el ajuste automático, que pueden crear recomendaciones de índice y mucho más. Es tentador hacer clic en Aplicar y tener algunas optimizaciones. No obstante, relájese ... ¡recuerde que no tendrá este cambio en otros entornos! Examine la recomendación, copie el SQL sugerido e introduzca los cambios mediante migraciones. Paso a paso, intente imitar su entorno de producción tanto como sea posible.
5. **Utilice el migrador de su elección para implementar su Base de Datos en todas partes.** Una vez que decida usar migraciones, úselas comenzando desde su máquina local y terminando en producción. Lamentablemente, los entornos son diferentes puesto que las configuraciones de Base de Datos en Producción no son las mismas que en QA, UAT o en el equipo local. Eso es lo que siempre vamos a enfrentar y tener esto en cuenta.
6. **No pierda su tiempo escribiendo migraciones.** Había estado haciendo esto durante mucho tiempo. Créame, no vale la pena. No corrí la migración ni una sola vez en mi vida en producción. Estoy totalmente de acuerdo con algunas personas que dicen que anotar las migraciones es una gran sobrecarga para el equipo y, sobre todo, será más rápido simplemente acumular una corrección de errores o, en el peor de los casos, restaurar una copia de seguridad.
7. **Las migraciones al inicio de la aplicación son una mala idea.** Como desarrollador .NET, admito que EntityFramework es un gran componente para creación y migración de Base de Datos, pero también, hay fanáticos aferrados a esta tecnología que piensan que con poner esta aplicación al inicio de un proyecto de .NET o .NET Core es la garantía absoluta de que *todo va a jalar bien sin preocuparnos del esquema de Base de Datos destino*. Los argumentos que le dirán estos fanáticos son:
 - **Razones de seguridad.** El usuario de la base de datos de su aplicación no debería tener derecho a crear o eliminar un objeto de base de datos (a menos que esté haciendo algo más específico, en la mayoría de los casos no es así). Con solo leer o escribir debería ser suficiente (error, del tipo epic fail).
 - **Escalado.** Imagine que desea implementar su aplicación en 5 instancias. Ahora tiene que lidiar con 5 procesos que ejecutan simultáneamente migraciones en una base de datos en lugar de una (pon tu santo al revés para que todo te salga bien).
 - Podrá implementar su aplicación con migraciones que no se pueden ejecutar correctamente. Esto simplemente hará que la aplicación caiga. Cuando las migraciones están separadas, primero puede implementar las migraciones y, si este paso fue exitoso, la aplicación. Puede ignorar esos errores al iniciar la aplicación, pero esta es una forma de perder el control.
 - **Introducirá el acoplamiento mental.** Después de un tiempo, todos asumirán que el nuevo código solo se ejecuta con el esquema de base de datos más reciente. Esto puede ser perjudicial si desea considerar la implementación azul / verde o las versiones canarias. En este enfoque, su aplicación debería poder funcionar con la versión de esquema anterior.
- Entonces... pues si no se consideran estos aspectos, por más bueno que uno sea usando EF Migrations, tendrá dolores de cabeza peores cuando se hagan esos cambios en Producción, parando todo.
8. **Dockerize las migraciones.** No es algo necesario, pero esto puede aumentar la productividad y más cuando se suban contenedores Docker con precompilación y migración de Bases de Datos antes de cargar la aplicación en la nube como Azure o AWS. Esto es posible y se puede hacer, pero es lo más sano para evitarnos problemas en ambiente de Producción.

9. **No tenga miedo de hablar sobre migraciones con representantes comerciales o del negocio.** A veces, simplemente tiene que preguntar acerca de los valores predeterminados para las nuevas columnas, en el idioma de la gente de negocios, por supuesto. Por ejemplo:

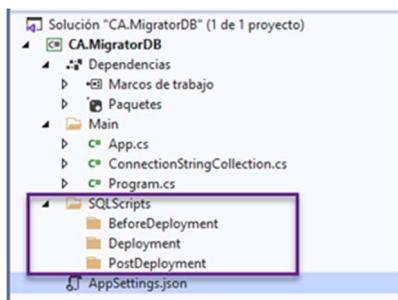
Oye, estoy terminando de agregar el número de cuenta a nuestros clientes, pero no sé qué deberíamos poner allí para los existentes. ¿Debería ser algún tipo de N/A que nuestros usuarios administradores deben completar más tarde o desea que los complete por adelantado con el archivo que me proporcionará?

10. **Los scripts de migración deben ser idempotentes.** Esto indica que los scripts SQL deben estar empaquetados con código adicional que verifique la existencia de los objetos de Base de Datos que se van a crear, modificar o eliminar. **Esto debe ser obligatorio**, independiente del motor de Base de Datos. En SQL Server siempre se usa esta buena práctica antes de hacer cambios en los objetos de Base de Datos. El migrador rastrea los scripts que ya se han ejecutado, pero cuando tiene idempotencia, puede cambiar fácilmente de un migrador a otro (ser independiente de una herramienta específica es una buena práctica). Simplemente mueva los scripts de migraciones a un nuevo proyecto y luego ejecútelo. La **idempotencia** es la propiedad para realizar una acción determinada varias veces y aun así conseguir el mismo resultado que se obtendría si se realizase una sola vez.

Siguiendo estas buenas sugerencias, seremos buenos en realizar procesos de migración de Bases de Datos.

1.6. Creando nuestra Base de Datos.

Para finalizar este capítulo, terminemos de completar nuestro proyecto de migración de Base de Datos, aplicando los consejos antes mencionados. Nos falta integrar los scripts SQL para realizar la migración a Base de Datos.



Vamos por partes:

1. En la carpeta **BeforeDeployment**, establecemos la regla de que se van a aplicar los comandos DDL (Data Definition Language) sobre la base de datos. Es decir: podemos definir los scripts DDL para crear usuarios, roles, asignar permisos de lectura y escritura a inicios de sesión de Base de Datos, etc. Nuestro primer script sería **[TimeStampUnix]-CreateSchema.sql**. El contenido de este archivo es el siguiente:

```
-- 1633584323-CreateSchema.sql
-- Autor: Olimpo Bonilla Ramírez.
-- Objetivo: Creación de un esquema de Base de Datos para objetos de Base de Datos, si no existen previamente.
-- Fecha: 2021-10-07.
-- Comentarios: Aquí se pueden crear en esta fase, los esquemas de Base de Datos con permisos sobre los objetos de Base de Datos.

IF NOT EXISTS ( SELECT * FROM sys.schemas t1 WHERE (t1.name = N'Sample') )
EXEC('CREATE SCHEMA [Sample] AUTHORIZATION [dbo];')
GO

-- 1633584323-CreateSchema.sql
```

Siempre creamos los scripts de esa manera poniendo al inicio y al final el nombre del archivo de script.

Debemos tambien poner el autor, correo electrónico y el propósito del script. Notese que es idempotente, por que estamos checando la existencia del objeto DDL en la Base de Datos, antes de crearlo, modificarlo o eliminarlo (en este caso, para Microsoft SQL Server, pero siempre hay que hacer esa comprobación, dependiendo del motor de Base de Datos).

2. **Analicemos la problemática.** Se crean las tablas de usuarios, sucursales y tipos de artículo. Eso no tenemos problema alguno en crear el script SQL de cada una de las tablas.

- o Creamos la tabla de usuarios con los campos siguientes:

Campo:	Descripción:
account_id	Identificador de la cuenta de usuario.
first_name	Nombre del usuario.
last_name	Apellidos del usuario.
username	Cuenta de usuario.
passwordhash	Contraseña (cifrada).
creationdate	Fecha de alta.
updatedate	Fecha de actualización.

- o Creamos la tabla de tipos de articulo.

Campo:	Descripción:
producttype_id	Identificador del tipo de producto.
description	Nombre del tipo de producto.
account_id	Identificador de la cuenta de usuario que hizo el alta del registro.
creationdate	Fecha de alta.
updatedate	Fecha de actualización.

- o Creamos la tabla de sucursales.

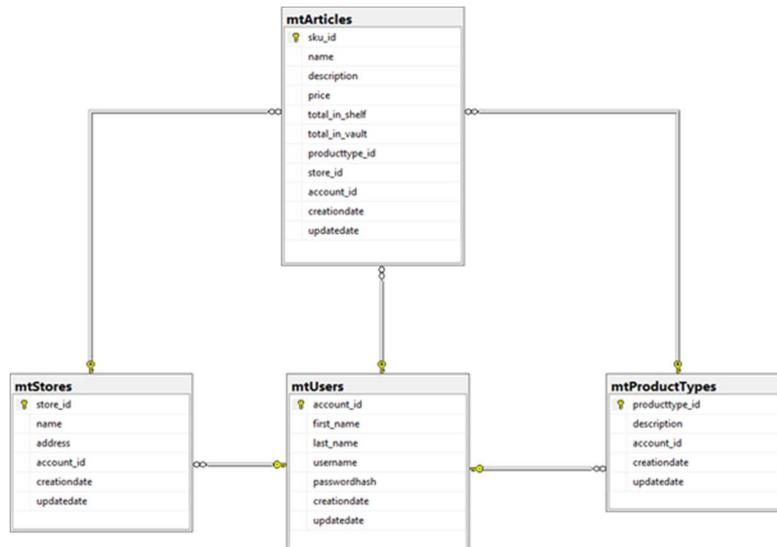
Campo:	Descripción:
store_id	Identificador de la sucursal.
name	Nombre de la sucursal.
address	Domicilio o dirección de la sucursal.
account_id	Identificador de la cuenta de usuario que hizo el alta del registro.
creationdate	Fecha de alta.
updatedate	Fecha de actualización.

- o Ahora, un artículo puede estar en varias sucursales y venderse en un precio unitario, según la sucursal. Entonces, crearemos la tabla de artículos con dos relaciones: uno a muchos con la tabla de Sucursales y uno a muchos con la tabla de Usuarios por que un vendedor captura un artículo nuevo. Tambien se hace una relación de uno a muchos con la tabla de tipos de artículo, esto para su clasificación. Su definición sería:

Campo:	Descripción:
sku_id	Identificador del artículo.
name	Nombre del artículo.
description	Descripción del artículo.
price	Precio unitario del artículo.
total_in_shelf	Total mínima de existencia en stock.
total_in_vault	Total máxima de existencia en stock.
producttype_id	Identificador del tipo de producto.

store_id	Identificador de la sucursal.
account_id	Identificador de la cuenta de usuario que hizo el alta del registro.
creationdate	Fecha de alta.
updatedate	Fecha de actualización.

Nuestro diagrama de entidad relación sería algo como esto:



3. Creamos las tablas del paso anterior, con el script siguiente en la carpeta **Deployment**, puesto que esta carpeta tiene como objetivo, ejecutar la creación de los objetos de Base de Datos y cargar los datos iniciales.

Tabla de cuentas de usuario **mtUsers**:

```

-- 1. Catálogo de cuentas de usuario.
-----

begin
drop table if exists dbo.mtArticles; drop table if exists dbo.mtProductTypes; drop table if exists dbo.mtStores; drop table if exists dbo.mtUsers;


print '<<< Ocurrió un error al eliminar el objeto ''dbo.mtUsers''. >>>';
else
print '<<< El objeto ''dbo.mtUsers'' se ha eliminado correctamente. >>>';
end;

create table dbo.mtUsers
(
[account_id] [int] IDENTITY(1, 1) NOT NULL,
[first_name] [varchar] (255) NOT NULL,
[last_name] [varchar] (255) NOT NULL,
[username] [varchar] (255) NOT NULL,
[passwordhash] [varchar] (255) NOT NULL,
[creationdate] [datetime] NOT NULL DEFAULT GETUTCDATE(),
[updatedate] [datetime] NULL,
CONSTRAINT [pk_IdUser] PRIMARY KEY (account_id),
CONSTRAINT [uq_IdUser] UNIQUE (account_id)
);
go

print '<<< El objeto ''dbo.mtUsers'' se ha creado correctamente. >>>';
else
print '<<< Ocurrió un error al crear el objeto ''dbo.mtUsers''. >>>';

  
```

Tabla de tipos de artículos, **mtProductTypes**.

```

-- 2. Catálogo de tipos de productos.

IF OBJECT_ID('dbo.mtProductTypes') IS NOT NULL
BEGIN
    DROP TABLE IF EXISTS dbo.mtArticles; DROP TABLE IF EXISTS dbo.mtProductTypes;

    IF OBJECT_ID('dbo.mtProductTypes') IS NOT NULL
        PRINT '<<< Ocurrió un error al eliminar el objeto ''dbo.mtProductTypes''. >>>';
    ELSE
        PRINT '<<< El objeto ''dbo.mtProductTypes'' se ha eliminado correctamente. >>>';

END;

CREATE TABLE dbo.mtProductTypes
(
    [producttype_id] [int] IDENTITY(1, 1) NOT NULL,
    [description] [varchar] (255) NOT NULL,
    [account_id] [int] NOT NULL,
    [creationdate] [datetime] NOT NULL DEFAULT GETUTCDATE(),
    [updatedate] [datetime] NULL,
    CONSTRAINT [pk_IdProductType] PRIMARY KEY (producttype_id),
    CONSTRAINT [uq_IdProductType] UNIQUE(producttype_id, account_id),
    CONSTRAINT [fk_IdProductType] FOREIGN KEY(account_id) REFERENCES mtUsers(account_id)
);
GO

IF OBJECT_ID('dbo.mtProductTypes') IS NOT NULL
    PRINT '<<< El objeto ''dbo.mtProductTypes'' se ha creado correctamente. >>>';
ELSE
    PRINT '<<< Ocurrió un error al crear el objeto ''dbo.mtProductTypes''. >>>';

```

Tabla de sucursales, **mtStores**.

```

-- 3. Catálogo de tiendas o sucursales.

IF OBJECT_ID('dbo.mtStores') IS NOT NULL
BEGIN
    DROP TABLE IF EXISTS dbo.mtArticles; DROP TABLE IF EXISTS dbo.mtStores;

    IF OBJECT_ID('dbo.mtStores') IS NOT NULL
        PRINT '<<< Ocurrió un error al eliminar el objeto ''dbo.mtStores''. >>>';
    ELSE
        PRINT '<<< El objeto ''dbo.mtStores'' se ha eliminado correctamente. >>>';

END;

CREATE TABLE dbo.mtStores
(
    [store_id] [int] IDENTITY(1, 1) NOT NULL,
    [name] [varchar] (255) NOT NULL,
    [address] [varchar] (255) NOT NULL DEFAULT 0,
    [account_id] [int] NOT NULL,
    [creationdate] [datetime] NOT NULL DEFAULT GETUTCDATE(),
    [updatedate] [datetime] NULL,
    CONSTRAINT [pk_IdStore] PRIMARY KEY (store_id),
    CONSTRAINT [uq_IdStore] UNIQUE(store_id, account_id),
    CONSTRAINT [fk_IdStore] FOREIGN KEY(account_id) REFERENCES mtUsers(account_id)
);
GO

IF OBJECT_ID('dbo.mtStores') IS NOT NULL
    PRINT '<<< El objeto ''dbo.mtStores'' se ha creado correctamente. >>>';
ELSE
    PRINT '<<< Ocurrió un error al crear el objeto ''dbo.mtStores''. >>>';

```

Tabla de artículos, **mtArticles**.

```

-- 4. Catálogo de artículos.
-----
IF OBJECT_ID('dbo.mtArticles') IS NOT NULL
BEGIN
    DROP TABLE dbo.mtArticles;

    IF OBJECT_ID('dbo.mtArticles') IS NOT NULL
        PRINT '<<< Ocurrió un error al eliminar el objeto ''dbo.mtArticles''. >>>';
    ELSE
        PRINT '<<< El objeto ''dbo.mtArticles'' se ha eliminado correctamente. >>>';

END;

CREATE TABLE dbo.mtArticles
(
    [sku_id]          [int]           IDENTITY(1, 1) NOT NULL,
    [name]            [varchar]      (255)          NOT NULL,
    [description]     [varchar]      (255)          NOT NULL DEFAULT 0,
    [price]           [money]        NOT NULL,
    [total_in_shelf] [int]          NOT NULL DEFAULT 0,
    [total_in_vault]  [int]          NOT NULL DEFAULT 0,
    [producttype_id] [int]          NOT NULL,
    [store_id]         [int]          NOT NULL,
    [account_id]       [int]          NOT NULL,
    [creationdate]    [datetime]     NOT NULL DEFAULT GETUTCDATE(),
    [updatedate]      [datetime]     NULL,
    CONSTRAINT [pk_Idarticle]
    PRIMARY KEY (sku_id),
    CONSTRAINT [uq_Idarticle] UNIQUE (sku_id, store_id),
    FOREIGN KEY(account_id) REFERENCES mtUsers(account_id),
    CONSTRAINT [fk_Idarticle1] FOREIGN KEY(store_id) REFERENCES mtStores(store_id),
    CONSTRAINT [fk_Idarticle2] FOREIGN KEY(producttype_id) REFERENCES mtProductTypes(producttype_id)
);
GO

IF OBJECT_ID('dbo.mtArticles') IS NOT NULL
PRINT '<<< El objeto ''dbo.mtArticles'' se ha creado correctamente. >>>';
ELSE
PRINT '<<< Ocurrió un error al crear el objeto ''dbo.mtArticles''. >>>';

```

Guardemos estos ajustes en el archivo **[TimeStampUnix]-CreateObjectsInit.sql**, el cual, indica la configuración de carga inicial de la Base de Datos. La carga de datos, la dejamos al gusto del usuario y esto se crea en el archivo **[TimeStampUnix]-LCAdTables.sql**.

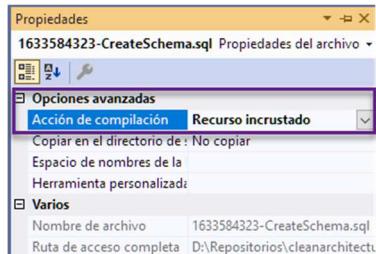
4. Por último, la carpeta **PostDeployment** es para probar que todos los componentes u objetos de Base de Datos creados anteriormente funcionen. Esta carpeta, a veces no la usamos, pero si debemos tener en cuenta que probando primero, asegura el éxito de la migración fluida en Base de Datos.
5. Finalmente, en el archivo del proyecto, incluyamos estas líneas, indicando que el contenido de la carpeta **SqlScripts** debe incluirse:

```

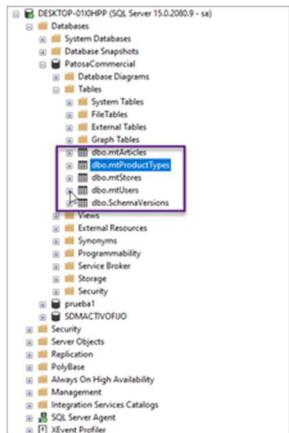
<!-- Scripts SQL. -->
<ItemGroup>
    <EmbeddedResource Include="SQLScripts\BeforeDeployment\*.sql" />
    <EmbeddedResource Include="SQLScripts\Deployment\*.sql" />
    <EmbeddedResource Include="SQLScripts\PostDeployment\*.sql" />
</ItemGroup>

```

6. Asegurarse que los archivos SQL estén como **Recurso incrustado (Embebed source)**:



Guardemos cambios y aplicamos la migración vemos que se han cargado las tablas en la Base de Datos con todo y sus datos. Revise las tablas y verifique el contenido de las mismas. Revisemos la Base de Datos y chequemos las tablas creadas:



Vemos que se ha creado una tabla llamada **SchemaVersions**, el cual, DbUp crea automáticamente para llevar el control de las migraciones aplicadas. Si revisamos esta tabla y corremos varias veces el archivo ejecutable, notamos que se ejecutó muchas veces un script... ¿Cuál de esos se ejecutará muchas veces cuando se haga una migración y en qué parte del deployment en Base de Datos ocurre esto? Lo dejó de tarea.

	Id	ScriptName	Applied
1	1	CA.MigratorDB.SQLScripts.BeforeDeployment.1633584323-CreateSchema.sql	2021-10-07 01:48:10.660
2	2	CA.MigratorDB.SQLScripts.Deployment.1633585172-CreateObjectsDb.sql	2021-10-07 01:48:10.747
3	3	CA.MigratorDB.SQLScripts.Deployment.1633588490-LoadTables.sql	2021-10-07 01:48:10.767
4	4	CA.MigratorDB.SQLScripts.PostDeployment.1633590409-CheckSP.sql	2021-10-07 02:09:16.657
5	5	CA.MigratorDB.SQLScripts.PostDeployment.1633590409-CheckSPsql	2021-10-07 02:09:29.883
6	6	CA.MigratorDB.SQLScripts.PostDeployment.1633590409-CheckSP.sql	2021-10-07 02:10:17.470

Con esto, terminamos nuestro proyecto de migración de Base de Datos, esencial para nuestro tutorial, en los capítulos siguientes.

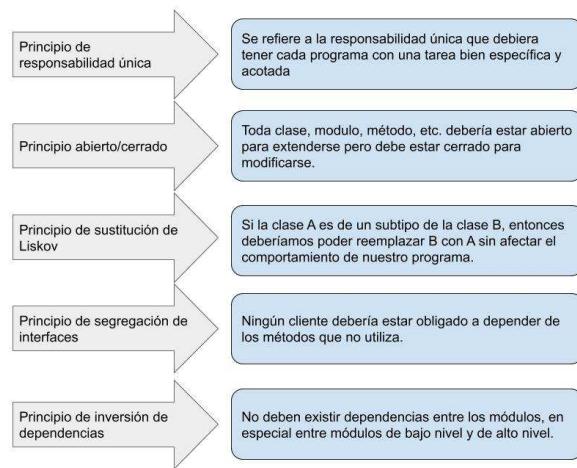
2. Clean Architecture.

En esta sección, veremos a manera detallada y sencilla el concepto de *Clean Architecture* y la manera de armar una plantilla que use esta arquitectura de software en Visual Studio. Tambien explicaremos en breve, los principios de SOLID para hacer un mejor código fuente y al final, generar el proyecto en Visual Studio con los principios antes mencionados.

2.1. Principios SOLID.

Los principios SOLID son una serie de recomendaciones para que podamos escribir un mejor código que nos ayude a implementar una alta cohesión y bajo acoplamiento.

Implementar SOLID en los proyectos puede ser una tarea simple o compleja, todo esto dependerá de la práctica pero tenerlos en cuenta desde un comienzo será más fácil de trabajar. La idea es buscar un punto de equilibrio ya que tal vez no todo nuestro proyecto necesite de dichos principios. Cada principio de SOLID está compuesta por cada inicial de este y son:



- **S:** Single Responsibility Principle (SRP). Un módulo solo debe tener **un motivo para cambiar**, dicho de otra manera, una clase debería estar destinada a una única responsabilidad y no mezclar la de otros o las que no le incumben a su dominio.
- **O:** Open/Closed Principle (OCP). Un módulo o clase de un software solo debe estar **abierto para su extensión** pero **cerrado para su modificación**.
- **L:** Liskov Substitution Principle (LSP). Si una clase A es un subtipo de la clase B, entonces, deberíamos **poder reemplazar la clase B con A, sin afectar o alterar su comportamiento**. Dicho de otra manera, **una clase hija puede ser usada como si fuera una clase padre sin alterar su comportamiento**.
- **I:** Interface Segregation Principle (ISP). Muchas interfaces específicas son mejores que una sola interfaz. En pocas palabras, **ningún cliente debería estar obligado a depender de los métodos que no utiliza**.
- **D:** Dependency Inversion Principle (DIP). Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones. En pocas palabras, **no deben existir dependencias entre módulos, en especial entre módulos de bajo nivel y de alto nivel**.

No explicaremos a detalle estos principios, por lo que pasaremos el siguiente punto.

2.2. Inyección de Dependencias: conceptos y definiciones.

Lo siguiente es fundamental para el desarrollo de aplicaciones en .NET Core y es el concepto de *inyección de dependencias*.

La inyección de dependencias es un conjunto de técnicas destinadas a disminuir el acoplamiento entre los componentes de una aplicación. Normalmente se le denomina DI que viene de sus siglas en inglés de Dependency Injection.

Cuando tenemos un objeto que necesita de otro para funcionar correctamente, tenemos definida una dependencia. Esta dependencia puede ser altamente acoplada o levemente acoplada. Si el acoplamiento es bajo el objeto independiente es fácilmente cambiante; si por el contrario es altamente acoplado, el reemplazo no es fácil y dificulta el diseño de los tests.

La inyección de dependencias es una metodología utilizada en los patrones de diseño que consiste en especificar comportamientos a componentes.

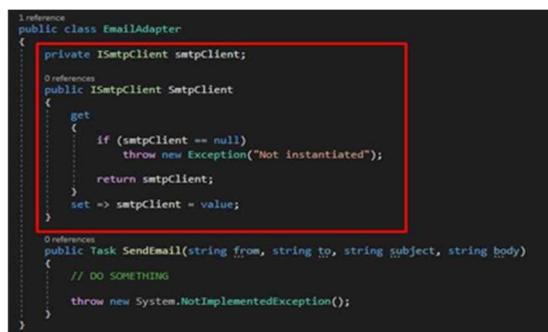
Se trata de extraer responsabilidades a un componente para delegarlas a otros componentes, de tal manera que cada componente solo tiene una responsabilidad (Principio de Responsabilidad Única de SOLID). Estas responsabilidades pueden cambiarse en tiempo de ejecución sin ver alterado el resto de comportamientos.

Ventajas de la inyección de dependencias. A continuación mencionaré, en mi opinión personal, algunas de las ventajas de usar DI en nuestras aplicaciones:

- Una de las grandes ventajas es el bajo acoplamiento entre los componentes, lo cual es una gran ayuda sobretodo en la mantenibilidad del software.
- **Es un patrón de diseño ampliamente conocido**, por ende es fácil de adaptar en múltiples lenguajes de programación, y son soluciones ya probadas a problemas recurrentes en el desarrollo de software.
- **Facilidad para pruebas**, ya que al tener componentes más desacoplados estos son más independientes, y como consecuencia se hace más fácil la implementación de prácticas recomendables de usar como TDD (Test Driven Development).
- **El software se hace más mantenable a medida que va creciendo**, ya que si se implementa una buena arquitectura con DI, la responsabilidad de cada uno de los componentes será muy clara y un cambio podrá ser más fácil de implementar.
- Al usar DI debemos pensar un poco más y planear mejor las dependencias de una clase, ya que la idea es utilizar solo las que sean necesarias.

Tipos de Inyección de Dependencias. En general, en mi opinión personal, considero yo los tipos de inyección de dependencias y estos son:

- **Por propiedad o atributo (Getter and/or Setter).** Ocurre cuando existe una clase o abstracción expuesta como propiedad o atributo.



```
1 reference
public class EmailAdapter
{
    private ISmtpClient smtpClient;

    public ISmtpClient SmtpClient
    {
        get
        {
            if (smtpClient == null)
                throw new Exception("Not instantiated");

            return smtpClient;
        }
        set => smtpClient = value;
    }

    public Task SendEmail(string from, string to, string subject, string body)
    {
        // DO SOMETHING

        throw new System.NotImplementedException();
    }
}
```

- **Por constructor (Constructor).** Cuando el constructor de una clase recibe una abstracción de otra clase como parámetro. Es el más comúnmente usado.

```
2 references
public class EmailAdapter
{
    private readonly ISmtpClient _smtpClient;

    0 references
    public EmailAdapter(ISmtpClient smtpClient) =>
        _smtpClient = smtpClient;

    0 references
    public Task SendEmail(string from, string to, string subject, string body)
    {
        // TODO SOMETHING

        throw new System.NotImplementedException();
    }
}
```

- **Por interfaz (Interface).** Cuando la clase tiene un método que recibe una abstracción por parámetro.

```
1 reference
public class EmailAdapter
{
    private ISmtpClient _smtpClient;

    0 references
    public void SetSmtpClient(ISmtpClient smtpClient) =>
        _smtpClient = smtpClient;

    0 references
    public Task SendEmail(string from, string to, string subject, string body)
    {
        // DO SOMETHING

        throw new System.NotImplementedException();
    }
}
```

- **Localizador de servicios (Service Locator).** A menudo se le llama **Contenedor**. Ocurre cuando tiene un contenedor propio que proporciona muchas instancias y solo puede solicitarle una instancia específica.

Para desarrollar este tipo de inyección de dependencias veamos lo siguiente paso a paso: el primero es crear una clase para registrar y proporcionar instancias.

```
0 references
public static class ServiceLocators
{
    private static IDictionary<string, Object> services = new Dictionary<string, Object>();

    0 references
    public static T Get<T>(string id) =>
        (T)services[id];

    0 references
    public static bool Has(string id) =>
        services.ContainsKey(id);

    0 references
    public static void Register<T>(string id, T service) =>
        services.Add(new KeyValuePair<string, Object>(id, service));
}
```

Después, en otra parte del código, registrar todas las instancias.

```
0 references
public void RegisterInstances()
{
    ServiceLocators.Register<ISmtpClient>("smtpClient", new SmtpClient());
```

Finalmente, podemos usarlo así:

```

1 reference
public class EmailAdapter
{
    private ISmtpClient _smtpClient;

    References
    public void setSmtpClient()
    {
        if (ServiceLocators.Has("smtpClient"))
            _smtpClient = ServiceLocators.Get<ISmtpClient>("smtpClient");
        else
            throw new Exception("Not instantiated");
    }

    References
    public Task SendEmail(string from, string to, string subject, string body)
    {
        // DO SOMETHING
        throw new System.NotImplementedException();
    }
}

```

Teniendo esto, pasaremos al siguiente apartado, para comprender el contenedor de inyección de dependencias.

2.3. Inversión de Control (IoC).

La **inversión de control**, o **IoC**, que es más conocido, es un patrón de diseño. Es una forma diferente de manipular el control de los objetos. Por lo general, depende de la inyección de dependencia, porque la creación de instancias de un objeto se convierte en una responsabilidad de la arquitectura, no del desarrollador.

Para poder utilizar inyección de dependencias, debemos indicar a qué clase hace referencia cada una de nuestras interfaces que estamos inyectando. Debemos configurar el contenedor de inyección de dependencias **cuando la aplicación comienza su ejecución**. Esto es por regla general en .NET Core.

En .NET Core utilizamos **IServiceCollection** para registrar nuestras propios servicios (dependencias). Cualquier servicio que quieras inyectar debe estar registrado en el contenedor **IServiceCollection** y estos serán resueltos por el tipo **IServiceProvider** una vez nuestro **IServiceCollection** ha sido construido.

La ubicación más común para registrar el contenedor es en el método **ConfigureServices** de la clase **Startup** en cualquier aplicación de .NET Core, tal como lo podemos ver en la siguiente imagen:

```

1 referencia | Olimpo Bonilla Ramírez, Hace 4 días | 1 autor, 1 cambio
static async Task Main(string[] args)
{
    var services = ConfigureServices();
    var serviceProvider = services.BuildServiceProvider(); ← IServiceProvider
    await serviceProvider.GetServiceApp().RunAsync(args);
}

1 referencia | Olimpo Bonilla Ramírez, Hace 4 días | 1 autor, 1 cambio
public static IConfiguration LoadConfiguration()
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("AppSettings.json", optional: true, reloadOnChange: true);
    return builder.Build();
}

1 referencia | Olimpo Bonilla Ramírez, Hace 4 días | 1 autor, 1 cambio
private static IServiceProvider ConfigureServices() ← IServiceProvider
{
    IServiceProvider services = new ServiceCollection();

    var config = LoadConfiguration();
    services.AddSingleton(config);

    /* Lectura de opciones del archivo de configuración. */
    services.Configure<ConnectionStringCollection>(options => config.GetSection($"CollectionConnectionStrings").Bind(options));

    /* Injectamos la clase 'App' */
    services.AddSingleton<App>();

    /* Otros servicios de la aplicación de la consola. */
    //services.AddTransient<IUser, User>();

    return services;
}

```

La manera de implementarlos es la siguiente, en el método antes mencionado:

```

services.AddTransient<IPersonalProfileInfo, PersonalProfileInfo>();
services.AddScoped<IPersonalProfileInfo, PersonalProfileInfo>();
services.AddSingleton<IPersonalProfileInfo, PersonalProfileInfo>();

```

Esto indica lo siguiente:

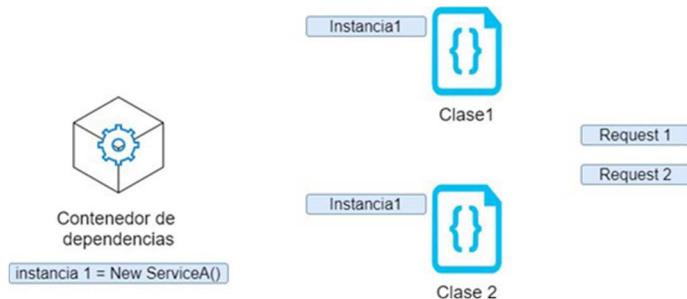
- El primer argumento es nuestra **abstracción**, o sea, la interfaz que vamos a utilizar a la hora de inyectar en los diferentes constructores.

- El segundo elemento es la **implementación** que hace uso de esa abstracción.
- En caso de que NO queramos introducir una abstracción, podemos indicar directamente la clase.

```
services.AddTransient<PersonalProfileInfo>();
```

Ciclo de vida de IoC. La inyección de dependencias tiene un ciclo de vida, por lo que elegir el adecuado para cada inyección de dependencias en el IoC es responsabilidad del desarrollador, ya que no todos los tipos de ciclo de vida traen los mismos resultados. .NET Core proporciona los tres tipos de inyección de dependencias, según su ciclo de vida.

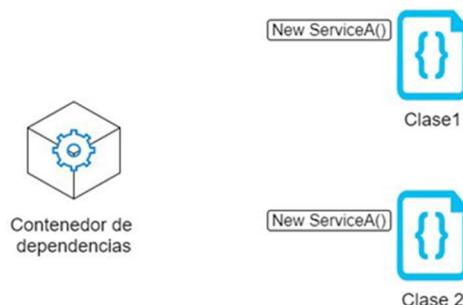
- **Scoped.** Los servicios se crearan **una vez** por solicitud del cliente.



El contenedor de IoC será el encargado de crear una instancia del tipo de servicio indicado por cada petición (request), siendo compartida esta instancia a lo largo de la petición (request).

```
services.AddSingleton<ILog, MyLog>();
/* O bien... */
services.AddSingleton(typeof(ILog), typeof(MyLog));
```

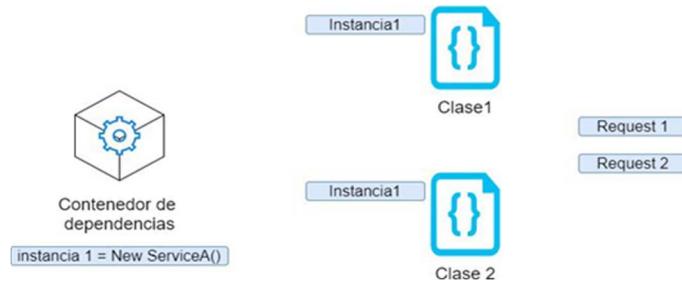
- **Trascient.** Los servicios se crearan **cada vez** que se resuelva la dependencia.



El contenedor de IoC será el encargado de crear una nueva instancia del tipo de servicio indicado cada vez que lo solicitemos.

```
services.AddTransient<ILog, MyLog>();
/* O bien... */
services.AddTransient(typeof(ILog), typeof(MyLog));
```

- **Singleton.** Los servicios que se crearán solo **la primera vez** que se resuelva la dependencia.

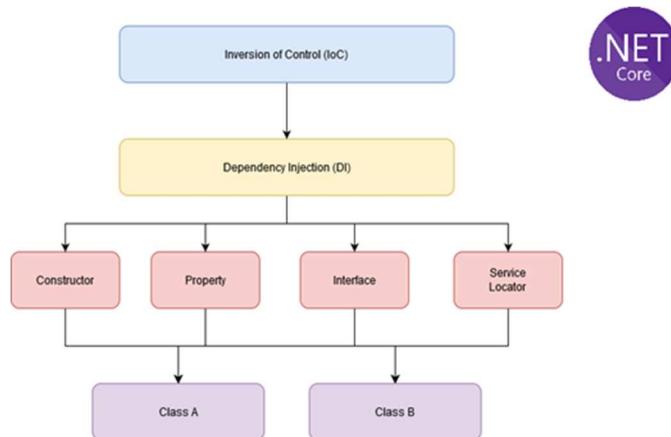


El contenedor de IoC será el encargado de crear y compartir una única instancia del servicio durante el funcionamiento de la aplicación en .NET Core.

```
services.AddSingleton<ILog, MyLog>();

/* O bien...
services.AddSingleton(typeof(ILog), typeof(MyLog));
```

El siguiente diagrama, resume en general, el modelo del pipeline de una aplicación en .NET Core, el cual, debemos tener en claro para futuros proyectos.



Consejos importantes. Como consejo, tomemos en cuenta esto:

- Hacer la inyección de dependencias por Constructor, en lugar de hacerlo por Getter or Setter o las demás tipos de inyección de dependencias.
- Implemente la interfaz `IDisposable`, para que ejecute automáticamente este método al intentar eliminar el servicio una vez concluido su ciclo de vida.
- Hacer pruebas unitarias para validar el comportamiento de los resultados por cada abstracción inyectada y decidir el ciclo de vida adecuado en el IoC para esa misma abstracción.

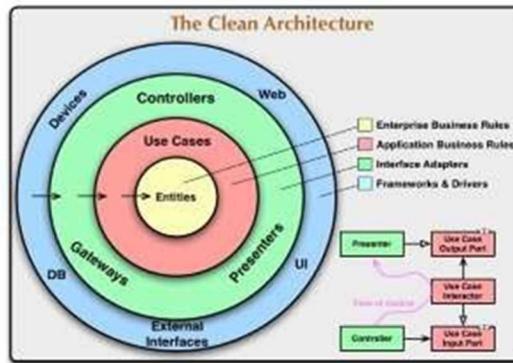
Con esto tenemos entonces comprendido el concepto de Inyección de Dependencias e Inversión de Control, para ya entrar de lleno al concepto de *Clean Architecture*.

2.4. Clean Architecture.

Este concepto fue desarrollado y planteado por **Robert C. Martin**, y dice así:

Clean Architecture (CA) es un conjunto de principios cuya finalidad principal es ocultar los detalles de implementación a la lógica de dominio de la aplicación.

De esta manera, mantenemos aislada la lógica, consiguiendo tener una lógica mucho más fácil de mantener y escalable en el tiempo. La representación de este modelo es el siguiente:



Lineamientos. Los lineamientos de esta arquitectura de software son los siguientes:

- La aplicación no debe depender de la interfaz de usuario, es decir que nuestra interfaz debería ser intercambiable (Windows Forms, Web, API, Móvil, etc.).
- La aplicación no debe depender de la base de datos (Oracle, SQL Server, MongoDB, etc.).
- Todas las capas deben poder probarse en forma independiente.
- No se debe depender de frameworks específicos, legacy systems, etc.

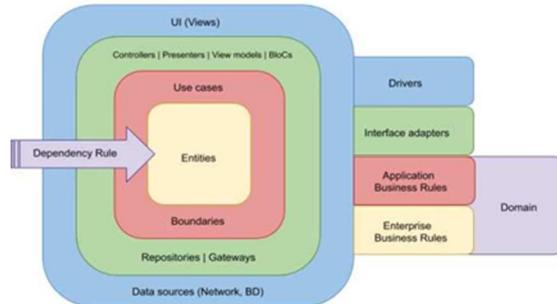
Regla de Dependencia. Para entender el diagrama tenemos que utilizar la regla de la dependencia:

"Las dependencias solo pueden apuntar hacia adentro. Nada en un círculo interno puede conocer algo de un círculo externo. El nombre de algo declarado en un círculo externo, no puede ser mencionado en un círculo interno".

O sea:

- La capa mas exterior representa los detalles de implementación.
- Las capas mas interiores representan el dominio, incluyendo lógica de aplicación y lógica de negocio empresarial.

La regla de dependencia nos dice que un círculo interior nunca debe conocer nada sobre un círculo exterior. Sin embargo, los círculos exteriores si pueden conocer círculos interiores.



Estructura general. Generalmente *Clean Architecture* se puede dividir en las siguientes capas:

- **Dominio.** Es el corazón de la aplicación y tiene que estar totalmente aislado de cualquier dependencia ajena a la lógica o datos del negocio.
- **Entidades.** Corresponden a los Objetos de Negocios que contienen datos y definen la conducta y reglas del negocio.
- **Casos de Uso.** Representan la lógica de la aplicación, que existe principalmente debido a la autorización de procesos mediante la aplicación y es inherente a cada aplicación.
- **Adaptadores (Repositories y Presenters).** Se encargan de transformar la información como se entiende y es representada en los detalles de la implementación.
- **Frameworks y Drivers.** En la capa más externa es donde van los detalles. Y la base de datos es un detalle, nuestro framework web, es un detalle etc. Aquí está la Base de Datos, el servicio al que alimentan los datos, etc.
- **Fronteras y límites (Boundary).** Una frontera es una separación que definimos en nuestra arquitectura para dividir componentes y definir dependencias. Estas fronteras tenemos que decidir dónde ponerlas, y cuándo ponerlas. Esta decisión es importante ya que puede condicionar el buen desempeño del proyecto. Una mala decisión sobre los límites puede complicar el desarrollo de nuestra aplicación o su mantenimiento futuro.

Por agrupación:

- **Dominio =>** Entidades y Casos de Uso.
- **Detalles de implementación.** => Frameworks y Drivers.
- **Adaptadores, Fronteras y Límites.**

Ventajas y desventajas de Clean Architecture. Como todo patrón de diseño, hay que exponer las ventajas y desventajas de Clean Architecture. Estas son:

Ventajas:

- Independencia de la interfaz de usuario.
- Las capas se prueban de manera independiente.
- El flujo de invocaciones es conocido y claro.
- Promueve buenas prácticas de Desarrollo.

Desventajas:

- Mayor número de clases e interfaces.
- El paso entre capas requiere un mapeo constante de los objetos creados.

¿Cuándo usar Clean Architecture? En general, si se decide usar este patrón de diseño, debe considerarse el uso. Tomemos en cuenta estas consideraciones.

Si aplica, cuando:

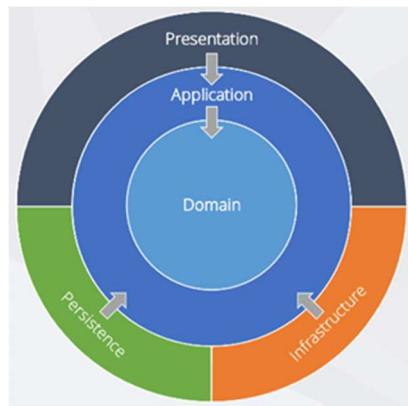
- El proyecto va a tener un tiempo de vida largo..
- Los objetos, reglas y conducta van a variar frecuentemente.
- Varios equipos de trabajo simultaneos.
- Un nivel alto de pruebas.

No aplica, cuando:

- Proyectos pequeños o de poco presupuesto.
- Si la aplicación es para un solo tipo de interfaz y no va a variar mucho.
- En aplicaciones de Reporting u operaciones CRUD básico.

2.5. Onion Architecture.

Una consecuencia del modelo de arquitectura de software de *Clean Architecture*, es sin duda el más conocido de todos: **Onion Architecture** (o *Arquitectura Cebolla*), el cual, se rige bajo el mismo principio de Clean Architecture, pero con algunas variantes.



Fue propuesto por en 2008, por **Jeffrey Palermo**, un poco antes que Robert C. Martin. Tambien se rige por los lineamientos antes mencionados en la sección anterior, y más que nada, en el principio de la regla de Dependencia, y de ventajas similares a *Clean Architecture*. Este modelo se divide en las siguientes capas:

Core o Domain. La capa más interna de este modelo. Esta capa no depende de las capas mas externas y contiene el modelo del negocio y aquí se encuentran las interfaces, entidades, modelos de dominio, interfaces de repositorio, etc. Estas interfaces incluyen abstracciones para las operaciones que se llevaran a cabo mediante la infraestructura, como el acceso a datos, etc. En ocasiones, los servicios o interfaces tendrán que trabajar con tipos sin entidad que no tienen dependencias en la interfaz de usuario o infraestructura, los cuales se llaman **Data Transfer Object (DTO)**. Los tipos son:

- Entities.
- Interfaces (de servicio y repositorio).
- Services.
- DTO.

Infrastructure. Es la siguiente capa superior a Core o Domain. Aquí incluyen las implementaciones de acceso a datos e implementaciones de servicios. Estos últimos tienen que interactuar con los intereses de la infraestructura y deben implementar interfaces definidas en la capa de Core, por lo que la infraestructura deberá tener una referencia a la capa de Domain. Los tipos de infraestructura son:

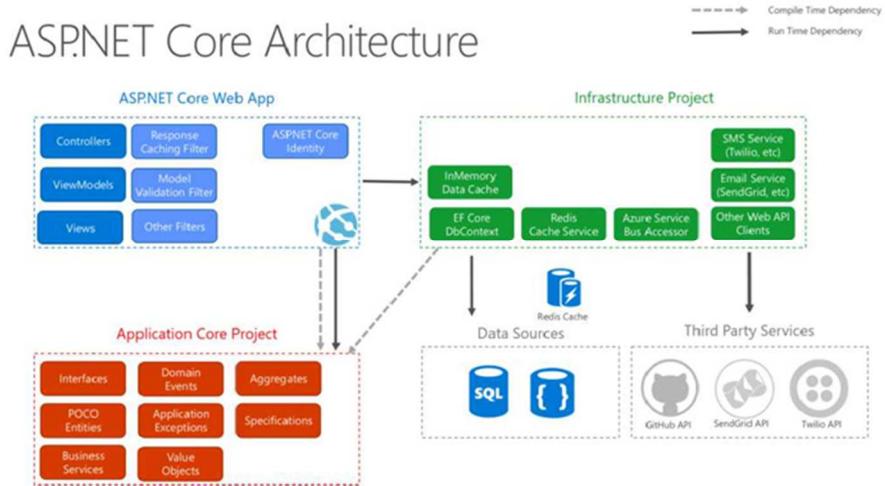
- Tipos de EntityFramework Core (DbContext, Migration).
- Tipos de implementación de acceso a datos (Repository).
- Servicios específicos de la infraestructura (por ejemplo FileLoggerService).

Interfaz de usuario (UI) o Presentación. Es el punto de entrada de la aplicación y la capa mas externa de esta arquitectura. Debe tener una referencia a las dos capas internas: Domain e Infrastructure. En esta capa, los consumidores pueden interactuar con los datos. Los tipos de esta capa son:

- Controllers.
- Filters.
- Views.
- ViewModels.
- Start.

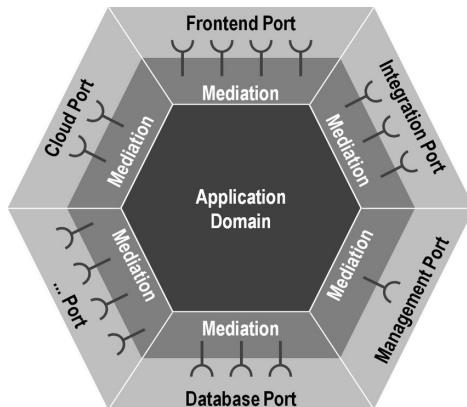
La clase **Startup** es responsable de configurar la aplicación y de conectar los tipos de implementación a las interfaces, lo que permite que la inserción de dependencias funcione correctamente en tiempo de ejecución.

Representando esto gráficamente sería esto:



2.6. Hexagonal Architecture.

Tambien es conocido como **Ports and Adapters**, y fue definido por **Alistair Cockburn** y adoptado por Steve Freeman y Nat Price en su libro *Growing Object Oriented Software*.



Este modelo, a veces, es poco usado, y tambien se rige bajo los mismos principios de *Clean Architecture*. Un ejemplo en .NET Core lo podemos encontrar [aquí](#).

2.7. Organizando nuestro proyecto.

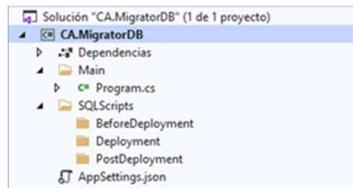
Una vez ya teniendo en claro estos conceptos, vamos entonces a implementar nuestro proyecto en Visual Studio. Necesitamos reorganizar nuestro proyecto inicial, puesto que hasta ahora, solo tenemos la aplicación de consola de Migrator DataBases (**CA.MigratorDB**). Entonces, pensando bien las cosas, decidimos usar el modelo *Onion Architecture* para nuestra problemática de este curso, con las siguientes observaciones:

- En un proyecto, unimos la capa de Application con Domain para armar una capa llamada Core. Esta es la capa mas interna.

- En otro proyecto, unimos la capa de Persistence con Infrastructure, para armar la capa llamada Infrastructure. Esta capa depende de Core.
- Finalmente crearemos la capa de User Interface, la cual, es la capa superior o la más externa y esta capa tomará como referencia las dos capas inferiores.

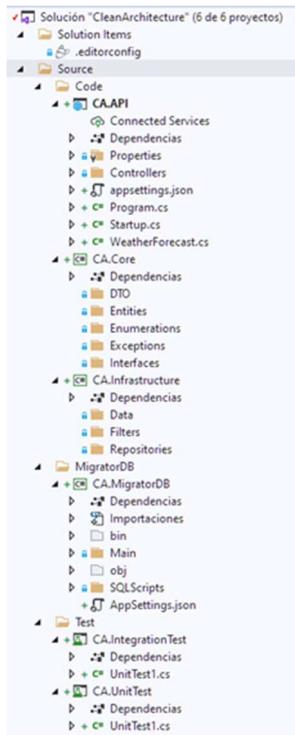
Entonces armaremos nuestro proyecto anterior como sigue:

1. Nuestro proyecto venia anteriormente así:



2. Quitemos el proyecto llamado **CA.MigratorDB** por un momento y abramos físicamente la carpeta de la solución **CleanArchitecture.sln**.
3. Creamos ahí tres nuevas carpetas llamadas **Test**, **MigratorDB** y **Code**. Pasemos el proyecto de migración de Base de Datos llamado **CA.MigratorDB** a la carpeta de **MigratorDB**.
4. Abramos Visual Studio y creamos tres nuevas carpetas de soluciones: **Test**, **MigratorDB** y **Code**.
5. Integremos de nuevo el proyecto **CA.MigratorDB** desde la nueva ubicación física donde se encuentra y pongamoslo en la carpeta de soluciones llamada **MigratorDB**.
6. Abramos dos nuevos proyectos del tipo librería: uno que se llama **CA.Core** y otro que se llama **CA.Infrastructure**. Estos proyectos hay que generarlos en la carpeta física llamada **Code** de la solución.
7. Abramos un nuevo proyecto del tipo **ASP.NET Core Web API** llamado **CA.Api**. Aquí habilitemos las opciones de configurar para peticiones **HTTPS** y la opción para *Habilitar para Docker*. Hay que crearlo físicamente en la carpeta de **Code**.
8. Ahora, asignemos las dependencias del proyecto: el proyecto **CA.Infrastructure** debe tener como referencia del proyecto a **CA.Core**. Y **CA.Api** debe tener como referencias de proyectos a **CA.Infrastructure** y a **CA.Core** respectivamente. **CA.Core no debe tener referencia de proyecto alguna** por que como sabemos, es la capa interna de la arquitectura de software.
9. Establezcamos a **CA.Api** como proyecto de arranque. Guardamos los cambios y compilamos.
10. Creamos en la carpeta de soluciones **Test** dos proyectos del tipo de pruebas unitarias del tipo XUnit. Hay que guardarlos en la carpeta física llamada **Test**. El primer proyecto es para tipo de pruebas unitarias llamado **CA.UnitTest** y el otro proyecto es para tipo de pruebas de integración llamado **CA.IntegrationTest**.
11. Ahora, en el proyecto **CA.Core**, creamos físicamente las carpetas siguientes: **DTO**, **Entities**, **Enumerations**, **Exceptions** e **Interfaces**. Conforme vayamos avanzando en este curso, crearemos las carpetas correspondientes a la capa más interna.
12. En el proyecto **CA.Infrastructure**, hay que crear las carpetas siguientes: **Data**, **Filters** y **Repositories**. Por el momento, empezaremos así y conforme vayamos avanzando, iremos creando más carpetas, apegado a los lineamientos de *Clean Architecture*.
13. Quitemos los archivos **Class1.cs** de los proyectos **CA.Core** y **CA.Infrastructure** respectivamente y compilemos de nuevo todo. Guardemos los cambios.
14. Si estamos desde un repositorio de git y tenemos el proyecto con un archivo llamado **.editorconfig**, agregarlo al proyecto. Creará una nueva carpeta llamada **Solution Items**.
15. Del paso anterior, crear una última carpeta de solución llamada **Source** y metamos las demás carpetas de soluciones en ella, menos la carpeta **Solution Items**.

Nuestro proyecto y su estructura debe quedar así.



Hay que asegurarnos que todos los proyectos de la carpeta **Code** y **MigratorDB** apunten a la plataforma de .NET Core 3.1 o superior preferentemente. Guardemos todo y compilemos. Todo debería de estar en orden y nos debería aparecer esto:

```
[{"date": "2021-10-13T20:21:06.2590436-05:00", "temperatureC": 19, "temperatureF": 66, "summary": "Mild"}, {"date": "2021-10-14T20:21:06.2628451-05:00", "temperatureC": -12, "temperatureF": 11, "summary": "Sweltering"}, {"date": "2021-10-15T20:21:06.2628515-05:00", "temperatureC": -10, "temperatureF": 15, "summary": "Balmy"}, {"date": "2021-10-16T20:21:06.262852-05:00", "temperatureC": 10, "temperatureF": 49, "summary": "Sweltering"}, {"date": "2021-10-17T20:21:06.2628524-05:00", "temperatureC": 40, "temperatureF": 103, "summary": "Freezing"}]
```

Lo cual indica que este proyecto está funcionando bien.

2.8. Creando el esqueleto de nuestra aplicación de Clean Architecture.

Para que todo esto funcione, faltan algunos retoques. Entonces vamos por ahora a crear los primeros módulos del proyecto en general, siguiendo los lineamientos de *Clean Architecture*.

Capa de Dominio (Domain Layer). Vamos a crear en la carpeta de **Entities**, las siguientes clases para las tablas de nuestra Base de Datos llamada **PatosaCommercial**.

- Ejecutemos el script que se encuentra en la carpeta **SQLScripts\PostDeployment** del proyecto **CA.MigratorDB** llamado **1634091036-SelectQueryGenerator.sql** en nuestra Base de Datos. Asignemos a las variables **@NombreTabla** y **@BaseDatos** los valores de **mtStores** y **PatosaCommercial** respectivamente. Ejecutemos el script y nos aparecerá algo como sigue:

2. Del set de datos resultante, copiemos la columna llamada **ColumnaCS**. Ahí es la definición de los atributos de la clase llamada **mtStores** en C#.
 3. En la carpeta **CA.Core\Entities** (haré así la ubicación de las carpetas de los proyectos de Clean Architecture de esta manera: nombre del proyecto\carpeta), creamos una clase llamada **mtStores** y copiemos la información de la columna del set de datos anterior para crear la entidad llamada **mtArticles** propiamente dicha. Nos debe quedar algo así como esto:

```
using System;

namespace CA.Core.Entities
{
    public class mStores
    {
        public int store_id { get; set; }

        public string name { get; set; }

        public string address { get; set; }

        public int account_id { get; set; }

        public DateTime creationdate { get; set; }

        public DateTime? updatedate { get; set; }
    }
}
```

4. Repitamos el mismo procedimiento para las demás tablas de la Base de Datos y guardemos los cambios y compilemos. No debería generarnos error alguno.

Capa de Infraestructura (Infrastructure Layer). Sigamos entonces con la capa de infraestructura, haciendo los siguientes pasos:

1. En la carpeta **CA.Infrastructure\Repositories** crear una clase llamada **StoreRepository.cs**. El contenido de este archivo es el siguiente.

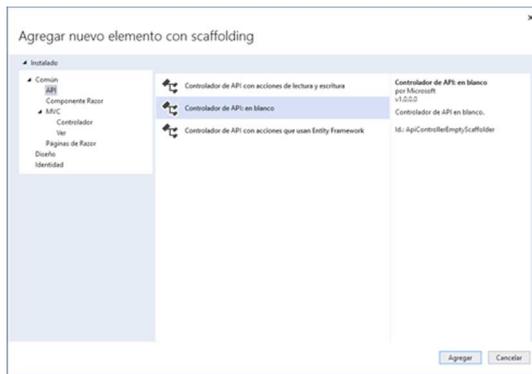
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using CA.Core.Entities;

namespace CA.Infrastructure.Repositories
{
    referencia | 0 cambios | 0 autores, 0 cambios
    public class StoreRepository
    {
        referencia | 0 cambios | 0 autores, 0 cambios
        public IEnumerable<mtStores> GetStores()
        {
            var _stores = Enumerable.Range(1, 50).Select(x => new mtStores
            {
                store_id = x,
                name = $"Store {x}",
                account_id = 1,
                address = $"Address for store {x}",
                creationdate = DateTime.UtcNow,
                updatedate = null
            });
            return _stores;
        }
    }
}
```

- Guardemos los cambios y compilamos. Para efectos de enseñanza, haremos datos dummy. Ya después ahora si nos conectaremos a la Base de Datos.

Capa de Presentacion (UI Layer). Finalmente completamos la capa de presentación, haciendo los siguientes pasos:

- En la carpeta **CA.Api\Controllers** crear un controlador llamado **StoreController.cs**. Lo podemos crear sobre la carpeta *Controllers* y pulsando el menú flotante, elegimos la ruta **Agregar\Controller** para que nos aparezca la siguiente ventana:



Elegimos *Controlador de API en blanco*, pulsamos *Agregar* y escribimos el nombre del controlador como **StoreController**. Pulsamos *Aceptar* y se generará el controlador en esa carpeta.

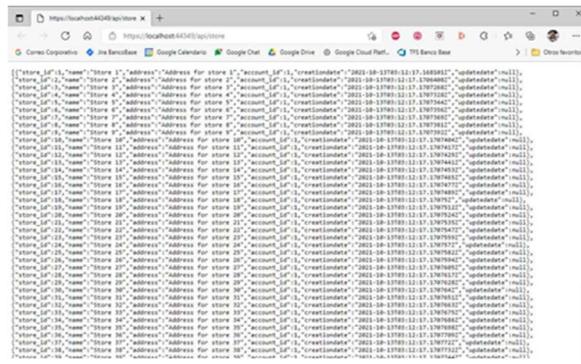
- Eliminemos el archivo **WeatherForecastController.cs** de esa carpeta antes mencionada.
- Eliminemos el archivo **WeatherForecast.cs** de la carpeta raíz del proyecto de **CA.Api**.
- Nuestro archivo llamado **StoreController.cs** debe tener algo como esto:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using CA.Infrastructure.Repositories;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace CA.Api.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class StoreController : ControllerBase
    {
        [HttpGet]
        public IActionResult GetStores()
        {
            var _stores = new StoreRepository().GetStores();
            return Ok(_stores);
        }
    }
}
```

Tal como se muestra en la imagen, agreguemos una función del tipo GET llamada **GetStores** con el código fuente de la imagen.

- Finalmente, en el archivo **launchSettings.json** de la carpeta **CA.Api\obj**, hay que cambiar el valor **launchUrl** por el valor **api\store** para que nuestro proyecto de presentación arranque, en tiempo de ejecución, en el controlador de **Stores**.
- Guardemos y compilemos. Si todo salió bien, debemos tener algo como esto:



Lo cual indica que hemos hecho esta plantilla de manera satisfactoria.

7. Probando en Postman o SCApUI, tendremos un resultado algo similar:

8. No olvidemos hacer referencias a las capas inferiores cada vez que creamos algún módulo nuevo en **Core**, **Infrastructure** y **Api** para que no tengamos errores de compilación.

Finalmente guardemos todo y vemos que ya tenemos nuestra plantilla de *Clean Architecture* lista para empezar a implementarse. Es importante aclarar que hay que empezar desde la capa mas interior y llegar a la capa mas alta, con el fin de que tengamos todo en orden y funcionando. En el siguiente capítulo, ya haremos el ajuste para que nuestro controlador traiga datos reales desde nuestra base de datos de PatosaCommercial, siguiendo los lineamientos de *Clean Architecture* explicados aquí.

3. Entity Framework Scaffold e Inyección de Dependencias.

En esta sección, veremos a manera detallada como usar el concepto de inyección de dependencias para hacer el proceso de Scaffolding de Entity Framework para nuestro proyecto de *Clean Architecture*, el cual, nos permita conectar a la Base de Datos de nuestro proyecto.

3.1. ¿Qué es Scaffold?

Scaffold es un método meta-programación de construcción de la base de datos backend de aplicaciones de software. Es una técnica apoyada por algunos modelos de controlador, en los que el programador puede escribir una especificación que describe cómo la base de datos de aplicación puede ser utilizada. El compilador utiliza esta especificación para generar el código que la aplicación puede utilizar para crear, leer, actualizar y eliminar las entradas de la base de datos, así también el tratamiento efectivo de la plantilla de diseño web como un "Scaffold" sobre la cual construir una aplicación potente y flexible.
El Scaffold es una evolución de los generadores de código de base de datos de los entornos de desarrollo anteriores, como caso de ejemplo el generador Entity Framework para SQL Server, y muchos otros productos de desarrollos de software como Dapper.

3.2. Especificaciones y nomenclatura para la Inyección de Dependencias.

Recordando el tema anterior de inyección de dependencias (DI) e inversión de control (IoC), vamos a definir el estandar de como se deben escribir los nombres de las abstracciones e implementaciones para el IoC que vamos a definir más adelante. Para nuestro problema, el contenedor IoC podría disponer de la siguiente nomenclatura de nombres de las clases e interfaces.

Abstracción:	Clase concreta:
<code>IArticleService</code>	<code>ArticleService</code>
<code>IArticleRepository</code>	<code>ArticleRepository</code>
<code>INotifier</code>	<code>EmailNotifier</code>

De esta forma, cuando una aplicación requiere una instancia de `IArticleService`, lo que haría, en lugar de crearla directamente, es solicitar al contenedor IoC un objeto `IArticleService`. Éste sabría que la clase concreta a crear es `ArticleService` y analizaría los parámetros de su constructor o propiedades decoradas con un atributo apropiado según el inyector que usemos.

En esta sección, vamos a usar estos principios antes mencionados, para la creación del modelo de Base de Datos con **EntityFramework.Core** en nuestro proyecto de *Clean Architecture*.

3.3. Scaffolding en .NET Core.

En el proyecto **CA.Infrastructure** se tiene que instalar los siguientes componentes de Nuget:

- [Microsoft.EntityFrameworkCore.](#)
- [Microsoft.EntityFrameworkCore.Relational.](#)
- [Microsoft.EntityFrameworkCore.Tools.](#)
- [Microsoft.EntityFrameworkCore.Abstractions.](#)
- [Microsoft.EntityFrameworkCore.Design.](#)

En el proyecto **CA.Api** se tiene que instalar los siguientes componentes de Nuget:

- [Microsoft.EntityFrameworkCore.Design.](#)

Considere lo siguiente:

1. Hay que asegurarse que los paquetes de EntityFrameworkCore coincidan en el número de versiones, en especial, *los cuatro primeros paquetes enlistados*.
 - Si se maneja el gestor de Base de Datos **Microsoft SQL Server**, tiene que agregarse el paquete de Nuget [Microsoft.EntityFrameworkCore.SqlServer.](#)
 - Si se usa **MySQL Server** o **MariaDB**, tiene que agregarse el paquete de Nuget [MySql.Data.EntityFrameworkCore](#) o [Pomelo.EntityFrameworkCore.MySql.](#)
 - Si se usa **PostgreSQL**, tiene que agregarse el paquete de Nuget [Npgsql.EntityFrameworkCore.PostgreSQL.](#)
2. Para generar el modelo de la Base de Datos, se necesita hacer un scaffolding por Entity Framework en la librería CA.Infrastructure, en la carpeta Data, por medio de la *Consola de Administrador de Paquetes de Visual Studio*.
 - Para **Microsoft SQL Server**:
`scaffold-dbcontext [CONNECTION_STRING] Microsoft.EntityFrameworkCore.SqlServer -OutputDir [OUTPUT_DIRECTORY] -Context [NAME_OF_CONTEXT_CLASS] -f`
 - Para **MariaDB** o **MySQL Server**:
`scaffold-dbcontext [CONNECTION_STRING] MySql.EntityFrameworkCore -OutputDir [OUTPUT_DIRECTORY] -Context [NAME_OF_CONTEXT_CLASS] -f`
`scaffold-dbcontext [CONNECTION_STRING] Pomelo.EntityFrameworkCore.MySql -OutputDir [OUTPUT_DIRECTORY] -Context [NAME_OF_CONTEXT_CLASS] -f`
 - Para **PostgreSQL**:
`scaffold-dbcontext [CONNECTION_STRING] Npgsql.EntityFrameworkCore.PostgreSQL -OutputDir [OUTPUT_DIRECTORY] -Context [NAME_OF_CONTEXT_CLASS] -f`

3. Se crearán los archivos de clases de contexto y del modelo de Base de Datos de Entity Framework Core en la carpeta Data con un nombre de contexto asignado a la implementación de Entity Framework asociado. Tambien se puede ejecutar desde la consola de Windows o Shell de Linux el siguiente comando, si se desea trabajar con *DataBase First*:

```
dotnet ef dbcontext scaffold [CONNECTION_STRING] [EF_COMPONENT] -OutputDir [OUTPUT_DIRECTORY] -Context [NAME_OF_CONTEXT_CLASS] -f
```

Si hay más cambios en Base de Datos para agregar nuevos objetos de BD, se repite el comando de scaffold por Entity Framework y se repiten los pasos mencionados en esta sección para su correcta ejecución.

3.4. Ajustes adicionales posterior a Scaffold.

Una vez realizado el proceso de scaffold de Entity Framework, el siguiente paso sería hacer ajustes adicionales para tener el código fuente lo mas corto posible. Para esto realizaremos los siguientes pasos.

1. Los archivos de clases de las entidades que se crearon en la carpeta **CA.Infrastructure\Data** se tienen que pasar a la carpeta **Entities** del proyecto **CA.Core** y cambiar su espacio de nombres correspondiente, que en este caso es **CA.Core.Entities**.
2. En el archivo de contexto generado, se tiene que eliminar la función llamada `onConfiguring` puesto que expone la cadena de conexión a la Base de Datos y *esto no es una buena práctica de programación*. La cadena de conexión se tomará desde la inyección de dependencias al arrancar el proyecto **CA.Api**.

3. Es necesario cambiar el nombre del archivo de contexto de Base de Datos por **DbContext.cs**, pulsando la combinación de teclas **Ctrl + RR**. Se cambiará tambien el nombre del archivo en la carpeta **Data** del proyecto **CA.Infrastructure**.
4. Incluir la siguiente línea de código al principio del archivo de contexto modificado:

```
using CA.Core.Entities;
```

5. En la carpeta **Repositories** del proyecto **CA.Infrastructure**, crear un archivo llamado **IEntityRepository.cs** y asocie la clase del tipo **EntityRepository.cs**, el cual, debe crearse en la carpeta **CA.Infrastructure\Repository**.

Por ejemplo, se crea una interfaz llamada **IArticleRepository.cs** en la carpeta **CA.Core\Interfaces**:

```
using System.Threading.Tasks;
using System.Collections.Generic;
using CA.Core.Entities;

namespace CA.Core.Interfaces
{
    public interface IArticleRepository
    {
        Task<IEnumerable<MtArticle>> GetArticles();
        Task<MtArticle> GetArticle(int id);
    }
}
```

Se crea una clase llamada **ArticleRepository.cs** en la carpeta **CA.Infrastructure\Repository** con la siguiente estructura:

```
using System;
using System.Linq;
using System.Collections;
using System.Threading.Tasks;
using System.Collections.Generic;

using CA.Core.Entities;
using CA.Core.Interfaces;
using CA.Infrastructure.Data;
using Microsoft.EntityFrameworkCore;

namespace CA.Infrastructure.Repositories
{
    public class ArticleRepository : IArticleRepository
    {
        private readonly PruebaContext _context;
        public ArticleRepository(PruebaContext pruebaContext) => _context = pruebaContext;

        public async Task<MtArticle> GetArticle(int id)
        {
            var _article = await _context.MtArticles.FirstOrDefaultAsync(x => x.IdArticle == id);
            return _article;
        }

        public async Task<IEnumerable<MtArticle>> GetArticles()
        {
            var _articles = await _context.MtArticles.ToListAsync();
            return _articles;
        }
    }
}
```

6. En cada archivo de código de CSharp, quite las referencias **using** innecesarias y guarde los cambios.
7. Si hay mas entidades para Entity Framework, repetir los pasos 3 al 6, creando cada repositorio a su entidad de Base de Datos correspondiente.
8. Compile los proyectos **CA.Core** y **CA.Infrastructure**. Si los pasos anteriores los hizo correctamente, no debería generarle errores de compilación.

3.5. Creación de Controllers.

Es necesario realizar la inyección de dependencias de cada entidad de Base de Datos en cada nuevo Controller. Esto lo haremos como sigue:

1. Para la entidad `mtArticle`, es necesario crear un archivo de controller llamado **ArticleController.cs** en la carpeta **CA.Api\Controllers**. Este archivo debe tener la siguiente estructura:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using CA.Core.Interfaces;

namespace CA.Api.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ArticleController : ControllerBase
    {
        private readonly IArticleRepository _articleRepository;
        public ArticleController(IArticleRepository articleRepository) => _articleRepository = articleRepository;

        [HttpGet]
        public async Task<IActionResult> GetArticles()
        {
            var _articles = await _articleRepository.GetArticles();
            return Ok(_articles);
        }

        [HttpGet("{id}")]
        public async Task<IActionResult> GetArticle(int id)
        {
            var _article = await _articleRepository.GetArticle(id);
            return Ok(_article);
        }
    }
}
```

2. Guarde los cambios y repita el paso anterior para referenciar de manera correcta el archivo del tipo Controller a las demás interfaces de entidades de Bases de Datos relacionadas y compilemos de nuevo el proyecto **CA.Api**.

3.6. Inyectando dependencias.

Para tener una mayor legibilidad del código fuente, es necesario realizar la inyección de todas las dependencias que hemos generado. Si nos hemos dado cuenta, desde las interfaces y controladores hemos visto algo como esto:

```
public ArticleController(IArticleRepository articleRepository) => _articleRepository = articleRepository;
```

En la práctica podemos **utilizar un sistema de inyección de dependencias**. Un sistema de inyección de dependencias es el encargado de instanciar las clases que necesitemos y suministrarnos ("inyectar") las dependencias enviando los parámetros oportunos al constructor.

3.7. Contenedor de Inversión de Control.

Ahora, tenemos que realizar la implementación del contenedor de inversión de control (**IoC Container**) a nivel general para que se haga uso de la abstracción de una instancia de clase en una clase y su implementación. Dicho de otra forma *resolvemos una abstracción para usarlo en una implementación*.

1. En el archivo **StartUp.cs** del proyecto **CA.Api**, busquemos la función `ConfigureServices` y escribamos la siguiente línea:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
```

```

/* Contenedor de inversión de control (IoC). */
services.AddTransient<IArticleRepository, ArticleRepository>();
}

```

Lo que estamos haciendo es realmente un contenedor de inversión de control (**IoC**), el cual, es un contenedor de inyección de dependencias y se puede separar y pasarlo en otro archivo de clase, el cual, lo haremos más adelante.

2. En el archivo **AppSettings.json** del proyecto **CA.Api**, tenemos que incluir la cadena de conexión a la Base de Datos, de la siguiente manera (esto depende del gestor de Base de Datos que esté usando en este proyecto):

```

{
  "ConnectionStrings": {
    "PruebaContext": "Server=localhost;Database=prueba1;Integrated Security=True;"
  }
}

```

3. Por ultimo, volvamos al archivo **StartUp.cs** y escribimos la siguiente línea de código en la función *ConfigureServices*:

```

public void ConfigureServices(IServiceCollection services)
{
  services.AddControllers();

  /* Cadena de conexión al contexto de Base de Datos. */
  services.AddDbContext<PruebaContext>(options => {
    options.UseSqlServer(Configuration.GetConnectionString("PruebaContext"));
  });

  /* Contenedor de inversión de control (IoC). */
  services.AddTransient<IArticleRepository, ArticleRepository>();
}

```

Lo que estamos haciendo aquí es que se lea desde el archivo de configuración, el valor de la cadena de conexión a la Base de Datos, y evitar su exposición en código fuente. En este caso usamos **UseSqlServer** por que estamos apuntando a Microsoft SQL Server. Para MySQL Server sería **UseMySQL** o para PostgreSQL sería **UsePostgreSQL**. Con esto, ya tenemos listo, a nivel de ámbito del proyecto, la cadena de conexión a Base de Datos y evitamos hacer esta definición de manera repetitiva en todo el código fuente.

4. Guardemos los cambios y compilemos **CA.Api** de nuevo. Notamos que ya podemos ejecutar la API REST de manera correcta y se nos muestra en el navegador la URL <https://localhost:44356/api/article> pero no le va a mostrar nada por que no tiene datos en la tabla **mtArticles** en Base de Datos.

4. Interfaz Fluida en Entity Framework.

En esta sección, veremos a manera detallada y sencilla como usar el concepto de **EF Fluent API (API Fluida para Entity Framework)** con el fin de configurar las clases de dominio para anular convenciones y tener listo todo para empezar a armar nuestro proyecto de *Clean Architecture* con las especificaciones y lineamientos de la arquitectura de software establecido.

4.1. Interfaz Fluida (Fluent Interface).

El concepto de **interfaz fluida (Fluent Interface)** es una construcción orientada a objetos que define un comportamiento capaz de retransmitir el contexto de la instrucción de una llamada subsecuente. Este término fue acuñado por primera vez por **Eric Evans y Martin Fowler**.

Generalmente, el contexto es:

- **Definido** a través del valor de retorno de un método llamado
- **Autoreferencial**, donde el nuevo contexto es equivalente al contexto anterior
- **Terminado** por medio del retorno de un contexto vacío (void context).

Este estilo es beneficioso debido a su capacidad de proporcionar una sensación más fluida al código, aunque algunos ingenieros encuentran el estilo difícil de leer. Una segunda crítica es que generalmente, las necesidades de programación son demasiado dinámicas para confiar en la definición estática de contacto ofrecida por una interfaz fluida.

En pocas palabras, una interfaz fluida proporciona una interfaz fácil de leer, que a menudo imita un lenguaje específico de dominio. El uso de este patrón da como resultado un código que se puede leer casi como lenguaje humano. El patrón de interfaz fluida es útil cuando desea proporcionar una API fluida y fácil de leer. Esas interfaces tienden a imitar lenguajes específicos de dominio, por lo que casi se pueden leer como lenguajes humanos.

Se puede implementar una interfaz fluida usando cualquiera de

- **Encadenamiento de métodos.** llamar a un método devuelve algún objeto en el que se pueden llamar a otros métodos.
- **Métodos e importaciones estáticos** de fábrica.
- **Parámetros con nombre**, que se pueden simular en Java utilizando métodos de fábrica estáticos.

Aplicando esto en el mundo real, necesitamos seleccionar números basados en diferentes criterios de la lista. Es una gran oportunidad para utilizar un patrón de interfaz fluido para proporcionar una experiencia de desarrollador legible y fácil de usar. En palabras sencillas, el patrón de interfaz fluida proporciona una interfaz fluida de fácil lectura para el código.

4.2. ¿Qué es Fluent API de Entity Framework?

EF Fluent API se basa en un patrón de diseño de **Fluent API** (también conocido como **Fluent Interface**) donde el resultado se formula mediante el **encadenamiento de métodos (method chaining)**.

En Entity Framework Core, la clase `ModelBuilder` actúa como una API fluida. Al usarlo, podemos configurar muchas cosas diferentes, ya que proporciona más opciones de configuración que el usar atributos de anotación de datos (*DataAnnotations*).

Entity Framework Core Fluent API configura los siguientes aspectos de un modelo:

- **Configuración del modelo.** Configura un modelo EF para asignaciones de base de datos. Configura el esquema predeterminado, las funciones de la base de datos, los atributos de anotación de datos adicionales y las entidades que se excluirán del mapeo.
- **Configuración de la entidad.** Configura el mapeo de la entidad a la tabla y las relaciones (orientado a llaves primarias, alternas, índices, nombre de la tabla, y las relaciones uno a uno, uno a muchos o muchos a muchos entre tablas, etc).
- **Configuración de propiedades.** Configura la propiedad a la asignación de columnas (nombre de las columnas, nulabilidad de las mismas, el tipo de dato, columna de simultaneidad, etc).

Teniendo esto concebido y entendido, empezamos entonces a ver estos puntos aplicados, a continuación.

4.3. Ajustando el idioma inglés.

Cuando hicimos el *scaffolding*, vimos que se generaron archivos de código: el principal es el archivo de contexto de Base de Datos y los archivos de entidades de Base de Datos. Esto se generó en la carpeta `CA.Infrastructure\Data`. Lo que tenemos que hacer ahora es lo siguiente:

1. **Pasar los archivos de entidades a la carpeta `CA.Core\Entities`.** Como nuestras tablas de migración del proyecto `CA.MigratorDB` están definidos los nombres de las tablas y sus columnas en inglés, y sus nombres están en notación *camelCase*, no hay necesidad de modificar la estructura de los datos. Si están los nombres de las tablas y columnas en otro idioma en los scripts de comandos DDL (Data Definition Language), es preferible dejarlos como están y seguir la especificación de como crear los objetos de Base de Datos en el idioma inglés o bien, aplicando la nomenclatura especificada. Una vez hecho este paso, cambiar el *namespace* de cada archivo de entidad de Base de Datos a:

```
namespace CA.Core.Entities
```

2. **Cambiar al idioma inglés.** Si los archivos de entidades tienen el nombre de la clase en un idioma distinto al inglés, hay que cambiarlos a su traducción **en inglés en singular**. Tomemos en cuenta esta siguiente estandar que nos puede servir para este caso:

Nombre original BD:	Nombre Scaffold:	Nombre correcto en inglés:
<code>mtArticulos</code>	<code>MtArticulos</code>	<code>Article</code>
<code>mtTiendas</code>	<code>MtTiendas</code>	<code>Store</code>
<code>mtTipCA.ctivo</code>	<code>MtTipCA.ctivo</code>	<code>AssetType</code>

Por ejemplo, veamos esta imagen siguiente de la entidad llamada `MtArticle` relacionada a la tabla `mtArticles` de Base de Datos:

MTArticle.cs Article.cs dbContext.cs AppSettings.json Startup.cs StoreController.cs
OA.Infrastructure

```
1 using System;
2 using System.Collections.Generic;
3
4 #nullable disable
5
6 namespace OA.Infrastructure.DataTmp
7 {
8     public partial class MTArticle
9     {
10         public int IdArticle { get; set; }
11         public string Name { get; set; }
12         public string Description { get; set; }
13         public decimal Price { get; set; }
14         public int TotalInShelf { get; set; }
15         public int TotalInVault { get; set; }
16         public int StoreId { get; set; }
17
18         public virtual MTStore Store { get; set; }
19     }
20 }
```

Remover esta línea.

Cambiar esto a inglés singular.

Si este atributo está en plural, cambiarlo a inglés singular.

Notamos que tiene el nombre de la clase llamada `MtArticle`. Está incorrecto, por que trae el nombre de la entidad desde la tabla de Base de Datos `mtArticle`. Entonces hay que cambiarlo a `Article`, pulsando las teclas **Ctrl + RR**. Se renombrará el nombre del objeto a lo largo del código fuente. Repitamos este procedimiento a los demás archivos de entidades de Base de Datos y guardemos los cambios en el proyecto **CA.Core**. Si encontramos alguna propiedad de la entidad con un nombre distinto al idioma inglés, *tambien hay que cambiarlo*.

3. **Usar nombres en plural.** Regresemos a la clase de contexto de Base de Datos y revisemos las siguientes líneas:

```
public virtual DbSet<MtArticle> MtArticles { get; set; }  
0 referencias | 0 cambios | 0 autores, 0 cambios  
public virtual DbSet<MtStore> MtStores { get; set; }  
0 referencias | 0 cambios | 0 autores, 0 cambios  
public virtual DbSet<SchemaVersion> SchemaVersions { get; set; }  
0 referencias | 0 cambios | 0 autores, 0 cambios  
public virtual DbSet<VwArticle> VwArticles { get; set; }
```

Vemos que los objetos `DbSet` están en inglés pero mal escritos por que no se aplica tampoco la notación `camelcase`. Aquí es bueno cambiarlos al inglés en plural. Por ejemplo: `MtArticles` sería `Articles` y así parea cada objeto `DbSet` correspondiente.

4. **Guardar cambios y compilamos.** Hasta aquí no debería generarnos error alguno. Aseguremonos que tanto los nombres de las entidades como sus atributos estén en el idioma inglés.

4.4. Configuración de entidades.

Si todo hemos seguido al pie de la letra, vemos que en nuestro archivo de contexto de Base de Datos, el método `OnModelCreating(ModelBuilder modelBuilder)` está un poco largo y extenso y se requiere simplificarlo, debido a que por la cantidad de entidades generadas desde la Base de Datos. La mejor práctica es segmentarlo en partes pequeñas para tener menos líneas de código.

```

0 referencias | 0 cambios | 0 autores, 0 cambios
protected override void OnModelCreating(ModelBuilder modelBuilder) Eliminar esta línea.
{
    modelBuilder.HasAnnotation("Relational:Collation", "Modern_Spanish_CI_AS");

    modelBuilder.Entity<MtArticle>(entity =>
    {
        entity.HasKey(e => new { e.IdArticle, e.StoreId })
            .HasName("pk_Idarticle");

        entity.ToTable("mtArticles");

        entity.HasIndex(e => new { e.IdArticle, e.StoreId }, "uq_Idarticle")
            .IsUnique();

        entity.Property(e => e.IdArticle)
            .ValueGeneratedOnAdd()
            .HasColumnName("id_article");

        entity.Property(e => e.StoreId).HasColumnName("store_id");

        entity.Property(e => e.Description)
            .IsRequired()
            .HasMaxLength(255)
            .IsUnicode(false)
            .HasColumnName("description")
            .HasDefaultValueSql("((0))");

        entity.Property(e => e.Name)
            .IsRequired()
            .HasMaxLength(255)
            .IsUnicode(false)
            .HasColumnName("name");

        entity.Property(e => e.Price)
            .HasColumnType("money")
            .HasColumnName("price");
    });
}

```

Para esto sigamos las siguientes instrucciones:

1. En la carpeta **CA.Infrastructure\Data**, crear una nueva carpeta llamada **Configurations**.
2. Dentro de la carpeta Configurations, crear una clase para cada entidad con el nombre de *EntidadConfiguration.cs*.

Por ejemplo: para la entidad Article el archivo sería ArticleConfiguration.cs y su estructura es la siguiente:

```

using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using CA.Core.Entities;

namespace CA.Infrastructure.Data.Configurations
{
    public class ArticleViewConfiguration : IEntityTypeConfiguration<Article>
    {
        public void Configure(EntityTypeBuilder<Article> builder)
        {
        }
    }
}

```

Notesé que estamos usando la interfaz heredable llamada **IEntityTypeConfiguration<Article>**, el cual, le estamos inyectando la clase de la entidad de Base de Datos llamada Article. Entonces solo nos falta mover las líneas de código fuente del método **onModelCreating** de la clase de contexto de Base de Datos que comprenden dentro de la definición **modelBuilder.Entity<Article>(entity => { ... })** al archivo que hemos creado, que es en este caso **ArticleConfiguration.cs**.

Cambiemos el texto **entity** por **builder**. Nos debería quedar algo como esto:

```

1  using OA.Core.Entities;
2  ...
3  using Microsoft.EntityFrameworkCore;
4  using Microsoft.EntityFrameworkCore.Metadata.Builders;
5
6  namespace OA.Infrastructure.Data.Configurations
7  {
8      ...
9      public class ArticleConfiguration : IEntityTypeConfiguration<Article>
10     {
11         ...
12         public void Configure(EntityTypeBuilder<Article> builder)
13         {
14             ...
15             builder.ToTable("mtArticles");
16
17             builder.HasKey(e => new { e.IdArticle, e.StoreId })
18                 .HasName("pk_Idarticle");
19
20             builder.HasIndex(e => new { e.IdArticle, e.StoreId }, "uq_Idarticle")
21                 .IsUnique();
22
23             builder.Property(e => e.IdArticle)
24                 .ValueGeneratedOnAdd()
25                 .HasColumnName("id_article");
26
27             builder.Property(e => e.StoreId).HasColumnName("store_id");
28
29             builder.Property(e => e.Description)
30                 .IsRequired()
31                 .HasMaxLength(255)
32                 .IsUnicode(false)
33                 .HasColumnName("description")
34                 .HasDefaultValueSql("((0))");
35
36         }
37     }
38 }

```

ESTO ES IMPORTANTE.
puesto que se va a mapear
desde la tabla de Base de
Datos.

Y repitamos lo mismo para las demás entidades de Base de Datos. Guardemos los cambios aplicados.

3. Regrese al archivo de contexto de Base de Datos y en el método `OnModelCreating` debe quedar algo como esto:

```

0 referencias | 0 cambios | 0 autores, 0 cambios
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
}

```

Solo nos falta incluir entonces la aplicación de la configuración de las entidades creadas, por lo que tenemos que escribir en `OnModelCreating`, lo siguiente:

```

0 referencias | Olimpo Bonilla Ramírez, Hace 1 día | 1 autor, 5 cambios
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new ArticleConfiguration());
    modelBuilder.ApplyConfiguration(new StoreConfiguration());
    modelBuilder.ApplyConfiguration(new ArticleViewConfiguration());
}

```

Pero si tenemos muchas entidades, tambien se tendría que escribir muchas líneas de código repitiendo el mismo procedimiento por cada entidad. Lo más fácil sería escribir esto:

```

0 referencias | Olimpo Bonilla Ramírez, Hace 1 día | 1 autor, 5 cambios
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // modelBuilder.ApplyConfiguration(new ArticleConfiguration());
    // modelBuilder.ApplyConfiguration(new StoreConfiguration());
    // modelBuilder.ApplyConfiguration(new ArticleViewConfiguration());

    modelBuilder.ApplyConfigurationsFromAssembly(typeof(PruebaContext).Assembly);
}

```

El cual, tomará todas las clases que hereden de la interfaz y se cargarán en el objeto `modelBuilder`.

4. Quitemos finalmente el método `OnModelCreatingPartial` del archivo de contexto y guardamos los cambios.
5. **Finalmente compilamos todo y probemos estos ajustes hechos.** Si todo salió bien, quiere decir que aplicamos de manera correcta el proceso de *Fluent API* para este proyecto. En caso de que salga algún error en tiempo de ejecución, significa que no hemos seguido los ajustes antes mencionados.

4.5. Quitando entidades innecesarias.

Es posible que dentro de la Base de Datos, existan tablas que no vamos a usar en nuestro archivo de contexto de Base de Datos. Lo más sano sería eliminar toda la referencia a las mismas y compilar todo. No pasa nada al respecto, pero si en algún futuro, se llegara a requerir, se puede aplicar el scaffolding de nuevo y ajustar esas nuevas entidades al estandar de *Fluent API* y haciendo los ajustes necesarios, apegados al estandar establecido.

Podemos quitar la entidad **SchemaVersions** el cual, se genera cuando hacemos migraciones a Bases de Datos por medio del proyecto **CA.MigratorDB** y se guardan en la tabla **SchemaVersions**. No la necesitamos, puesto que francamente esta tabla lleva el historico de las migraciones y cambios que hemos hecho a la Base de Datos y sin problema alguno, solo tomaremos las entidades para crear sus propias configuraciones de entidad e integrarlas a **CA.Core** y **CA.Infrastructure** respectivamente.

4.6. Resumen.

Definitivamente, en mi opinión personal, la técnica de API Fluida es esencial para tener un código fuente legible y mas fragmentado por que ayuda mucho al Desarrollador Senior a tener todo en orden y en partes pequeñas para su correcto funcionamiento y optimización. Cuando usemos Entity Framework, es importante que despues del proceso de scaffold hacer los ajustes necesarios en los objetos de entidad y sus relaciones para que funcione de manera correcta y óptima y seguir el estandar de optimización para esta funcionalidad.

5. Conceptos de HTTP y REST.

En esta sección, veremos algunas cosas de métodos HTTP para servicios RESTful. No ahondaremos en estos temas por que está fuera del alcance de este tutorial. Solo lo resumiremos de forma básica para tener un argumento teórico sobre el tema.

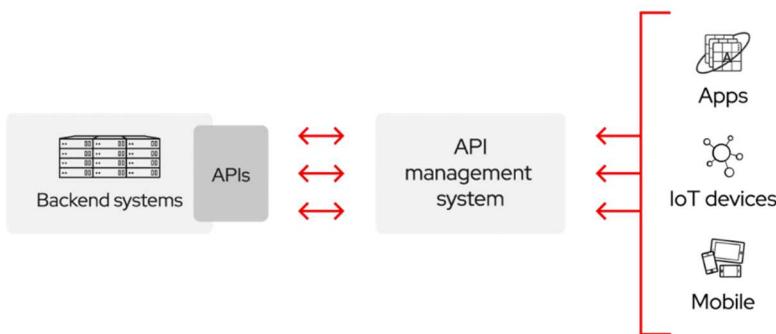
5.1. Concepto de API.

Una **API** es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones. API significa **interfaz de programación de aplicaciones**.

Las API permiten que sus productos y servicios se comuniquen con otros, sin necesidad de saber cómo están implementados. Esto simplifica el desarrollo de las aplicaciones y permite ahorrar tiempo y dinero. Las API le otorgan flexibilidad; simplifican el diseño, la administración y el uso de las aplicaciones, y proporcionan oportunidades de innovación, lo cual es ideal al momento de diseñar herramientas y productos nuevos (o de gestionar los actuales).

A veces, las API se consideran como contratos, con documentación que representa un acuerdo entre las partes: si una de las partes envía una solicitud remota con cierta estructura en particular, esa misma estructura determinará cómo responderá el software de la otra parte.

Las API son un medio simplificado para conectar su propia infraestructura a través del desarrollo de aplicaciones nativas de la nube, pero también le permiten compartir sus datos con clientes y otros usuarios externos. Las API públicas representan un valor comercial único porque simplifican y amplían la forma en que se conecta con sus partners y, además, pueden rentabilizar sus datos (un ejemplo conocido es la API de Google Maps).

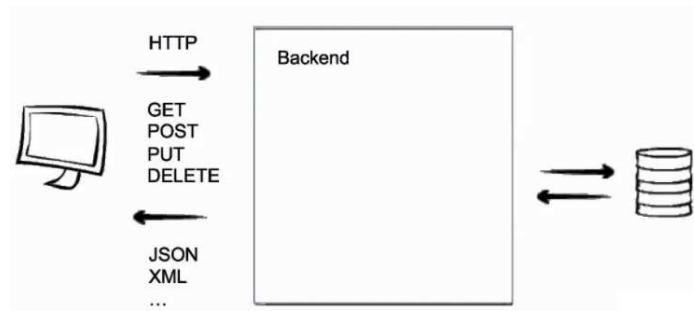


5.2. REST y API REST.

REST es una interfaz para conectar varios sistemas basados en el protocolo HTTP (uno de los protocolos más antiguos) y nos sirve para obtener y generar datos y operaciones, devolviendo esos datos en formatos muy específicos, como XML y JSON.

El formato más usado en la actualidad es el formato JSON, ya que es más ligero y legible en comparación al formato XML. Elegir uno será cuestión de la lógica y necesidades de cada proyecto.

REST se apoya en HTTP, del cual, surge como una alternativa a **SOAP**. Cuando hablamos de SOAP hablamos de una arquitectura dividida por niveles que se utilizaba para hacer un servicio, es más complejo de montar como de gestionar y solo trabajaba con XML.



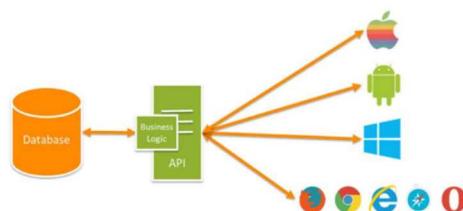
Ahora bien, REST llega a solucionar esa complejidad que añadía SOAP, haciendo mucho más fácil el desarrollo de una API REST, en este caso de un servicio en el cual nosotros vamos a almacenar nuestra lógica de negocio y vamos servir los datos con una serie de recursos URL y una serie de datos que nosotros los limitaremos, es decir, será nuestro **BACKEND** nuestra lógica pura de negocios que nosotros vamos a utilizar. Claro: no hablaré de SOAP tampoco, por que esto está fuera del alcance de este tutorial.

¿Por qué debemos utilizar REST? REST no es solo una moda, y es por las siguientes razones que esta interfaz está teniendo tanto protagonismo en los últimos años:

- **Crea una petición HTTP** que contiene toda la información necesaria, es decir, un REQUEST a un servidor tiene toda la información necesaria y solo espera una RESPONSE, ósea una respuesta en concreto.
- Se apoya sobre un protocolo que es el que se utiliza para las páginas web, que es HTTP, es un protocolo que existe hace muchos años y que ya está consolidado, no se tiene que inventar ni realizar cosas nuevas.
- **Se apoya en los métodos básicos de HTTP**, como son POST, GET, PUT, PATCH y DELETE (lo veremos más adelante).
- **Todos los objetos se manipulan mediante URI, por ejemplo**, si tenemos un recurso usuario y queremos acceder a un usuario en concreto nuestra URI sería `/user/identificadordelobjeto`, con eso ya tendríamos un servicio USER preparado para obtener la información de un usuario, dado un ID. URI es en realidad **un identificador de recursos**

Las ventajas de REST:

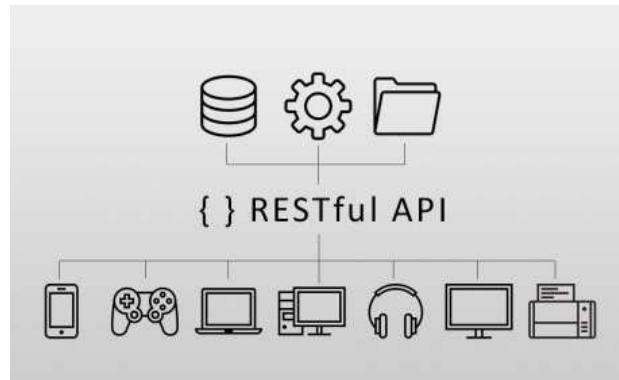
- Nos permite **separar el cliente del servidor**. Esto quiere decir que nuestro servidor se puede desarrollar en Node y Express, y nuestra API REST con Vue por ejemplo, no tiene por qué estar todos dentro de un mismo.
- Podemos crear un **diseño de un microservicio orientado a un dominio (DDD)**
- Es totalmente independiente de la plataforma, así que **podemos hacer uso de REST tanto en Windows, Linux, Mac** o el sistema operativo que nosotros queramos.
- Podemos hacer nuestra **API pública**, permitiendo darnos visibilidad si la hacemos pública.
- Nos da **escalabilidad**, porque tenemos la separación de conceptos de **CLIENTE** y **SERVIDOR**, por tanto, podemos dedicarnos exclusivamente a la parte del servidor.



Entonces, **API REST**, es una interfaz de programación de aplicaciones que se apoya en la arquitectura REST para el desarrollo de aplicaciones en red. Aprovechando el lenguaje HTML, permite que cualquier empresa cree aplicaciones web sin problemas, aunque siempre en base a las restricciones que supone.

5.3. RESTful.

En cambio, RESTful se basa en la tecnología de transferencia de estado representacional (**REST**), un estilo arquitectónico y un enfoque de las comunicaciones a menudo utilizadas en el **desarrollo de servicios web**.



5.4. CRUD.

El concepto **CRUD** está estrechamente vinculado a la gestión de datos digitales. CRUD hace referencia a un acrónimo en el que se reúnen las primeras letras de las cuatro operaciones fundamentales de aplicaciones persistentes en sistemas de bases de datos:

- **Create.** Crear registros.
- **Read bzw.** Retrieve (leer registros).
- **Update.** (Actualizar registros).
- **Delete bzw.** Destroy (Borrar registros).

En pocas palabras, CRUD resume las funciones requeridas por un usuario para crear y gestionar datos. Varios procesos de gestión de datos están basados en CRUD, en los que dichas operaciones están específicamente adaptadas a los requisitos del sistema y de usuario, ya sea para la gestión de bases de datos o para el uso de aplicaciones. Para los expertos, las operaciones son las herramientas de acceso típicas e indispensables para comprobar, por ejemplo, los problemas de la base de datos, mientras que para los usuarios, CRUD significa crear una cuenta (CREATE) y utilizarla (READ), actualizarla (UPDATE) o borrarla (DELETE) en cualquier momento. Dependiendo de la configuración regional, las operaciones CRUD pueden implementarse de diferentes maneras, como lo muestra la siguiente tabla:

CRUD-Operation	SQL	RESTful HTTP	XQuery
Create	INSERT	POST, PUT	insert
Read	SELECT	GET, HEAD	copy/modify/return
Update	UPDATE	PUT, PATCH	replace, rename
Delete	DELETE	DELETE	delete

5.5. HTTP Verbs.

Los verbos HTTP comprenden una parte importante de nuestra restricción de "interfaz uniforme" y nos proporcionan la acción equivalente al recurso basado en sustantivos. Los verbos HTTP principales o más utilizados (o métodos, como se les llama correctamente) son POST, GET, PUT, PATCH y DELETE. Corresponden a las operaciones de creación, lectura, actualización y eliminación (o CRUD), respectivamente. También hay otros verbos, pero se utilizan con menos frecuencia. De esos métodos menos frecuentes, OPTIONS y HEAD se utilizan con más frecuencia que otros.

A continuación se muestra una discusión más detallada de los principales métodos HTTP.

- **POST.** Este verbo se utiliza con mayor frecuencia para *crear* nuevos recursos. En particular, se usa para crear recursos subordinados. Es decir, subordinado a algún otro recurso (por ejemplo, padre). En otras palabras, al crear un nuevo recurso, POST al padre y el servicio se encarga de asociar el nuevo recurso con el padre, asignar un ID (nuevo URI de recurso), etc.

En la creación exitosa, devuelva el estado HTTP 201, devolviendo un encabezado de ubicación con un enlace al recurso recién creado con el estado HTTP 201.

POST no es ni seguro ni idempotente. Por lo tanto, se recomienda para solicitudes de recursos no idempotentes. Si hace dos solicitudes POST idénticas, lo más probable es que dos recursos contengan la misma información.

- **GET.** El método HTTP GET se utiliza para *leer* (o recuperar) una representación de un recurso. En la ruta "feliz" (o sin error), GET devuelve una representación en XML o JSON y un código de respuesta HTTP de 200 (OK). En caso de error, la mayoría de las veces devuelve un 404 (*Not Found*) o 400 (*Bad request*).

De acuerdo con el diseño de la especificación HTTP, las solicitudes GET (junto con HEAD) se utilizan solo para leer datos y no para cambiarlos. Por lo tanto, cuando se usan de esta manera, se consideran seguros. Es decir, se pueden llamar sin riesgo de modificación o corrupción de datos; llamarlo una vez tiene el mismo efecto que llamarlo 10 veces, o ninguno. Además, GET (y HEAD) es idempotente, lo que significa que hacer múltiples solicitudes idénticas termina teniendo el mismo resultado que una sola solicitud.

No exponga operaciones inseguras a través de GET. Como regla, nunca debemos modificar ningún recurso en el servidor.

- **PUT.** Este método se utiliza con mayor frecuencia para capacidades de *actualización*, a una URI de recurso conocido con el cuerpo de la solicitud que contiene la representación recién actualizada del recurso original.

Sin embargo, PUT también se puede utilizar para crear un recurso en el caso de que el cliente elija el ID del recurso en lugar del servidor. En otras palabras, si el PUT es para un URI que contiene el valor de un ID de recurso inexistente. Nuevamente, el cuerpo de la solicitud contiene una representación de recursos. Muchos sienten que esto es complicado y confuso. En consecuencia, este método de creación debe usarse con moderación, si es que se usa. Alternativamente, use POST para crear nuevos recursos y proporcione el ID definido por el cliente en la representación del cuerpo, presumiblemente a un URI que no incluye el ID del recurso.

En una actualización exitosa, devuelva 200 (o 204 si no devuelve ningún contenido en el cuerpo) de un PUT. Si usa PUT para crear, devuelva el estado HTTP 201 en la creación exitosa. Un cuerpo en la respuesta es opcional, siempre que uno consuma más ancho de banda. No es necesario devolver un enlace a través de un encabezado de ubicación en el caso de creación, ya que el cliente ya estableció el ID del recurso.

PUT no es una operación segura, ya que modifica (o crea) el estado en el servidor, pero es idempotente. En otras palabras, si crea o actualiza un recurso usando PUT y luego hace la misma llamada nuevamente, el recurso todavía está allí y aún tiene el mismo estado que tenía con la primera llamada.

Si, por ejemplo, llamar a PUT en un recurso incrementa un contador dentro del recurso, la llamada ya no es idempotente. A veces eso sucede y puede ser suficiente para documentar que la llamada no es idempotente. Sin embargo, se recomienda mantener las solicitudes PUT idempotentes. Se recomienda encarecidamente utilizar POST para solicitudes no idempotentes.

- **PATCH.** Este método se utiliza para *modificar* capacidades. La solicitud PATCH solo debe contener los cambios en el recurso, no el recurso completo. Esto se parece a PUT, pero el cuerpo contiene un conjunto

de instrucciones que describen cómo se debe modificar un recurso que reside actualmente en el servidor para producir una nueva versión. Esto significa que el cuerpo de PATCH no debería ser solo una parte modificada del recurso, sino en algún tipo de lenguaje de parche como JSON Patch o XML Patch.

PATCH no es seguro ni idempotente. Sin embargo, una solicitud de PATCH se puede emitir de tal manera que sea idempotente, lo que también ayuda a prevenir malos resultados de colisiones entre dos solicitudes de PATCH en el mismo recurso en un período de tiempo similar. Las colisiones de múltiples solicitudes de PATCH pueden ser más peligrosas que las colisiones PUT porque algunos formatos de parche necesitan operar desde un punto base conocido o de lo contrario dañarán el recurso. Los clientes que utilizan este tipo de aplicación de parche deben utilizar una solicitud condicional de modo que la solicitud falle si el recurso se ha actualizado desde la última vez que el cliente accedió al recurso. Por ejemplo, el cliente puede usar un ETag fuerte en un encabezado If-Match en la solicitud PATCH.

- **DELETE.** Esta función es bastante fácil de entender. Se utiliza para *eliminar* un recurso identificado por un URI. En caso de eliminación exitosa, devuelva el estado HTTP 200 (OK) junto con un cuerpo de respuesta, tal vez la representación del elemento eliminado (a menudo exige demasiado ancho de banda) o una respuesta ajustada (consulte Valores devueltos a continuación). O eso o devolver el estado HTTP 204 (SIN CONTENIDO) sin cuerpo de respuesta. En otras palabras, un estado 204 sin cuerpo, o la respuesta estilo JSEND y el estado HTTP 200 son las respuestas recomendadas.

En cuanto a las especificaciones HTTP, las operaciones DELETE son idempotentes. Si ELIMINA un recurso, se elimina. Llamar repetidamente a DELETE en ese recurso termina igual: el recurso se ha ido. Si llamar a DELETE digamos, disminuye un contador (dentro del recurso), la llamada DELETE ya no es idempotente. Como se mencionó anteriormente, las estadísticas de uso y las mediciones pueden actualizarse mientras se sigue considerando que el servicio es idempotente siempre que no se modifiquen los datos de los recursos. Se recomienda usar POST para solicitudes de recursos no idempotentes.

Sin embargo, hay una advertencia sobre ELIMINAR la idempotencia. Llamar a DELETE en un recurso por segunda vez a menudo devolverá un 404 (NO ENCONTRADO) ya que ya se eliminó y, por lo tanto, ya no se puede encontrar. Esto, según algunas opiniones, hace que las operaciones DELETE ya no sean idempotentes, sin embargo, el estado final del recurso es el mismo. Devolver un 404 es aceptable y comunica con precisión el estado de la llamada.

HTTP Method	Safe	Idempotent
GET	✓	✓
POST	✗	✗
PUT	✗	✓
DELETE	✗	✓
OPTIONS	✓	✓
HEAD	✓	✓

En resumen, esta es nuestra tabla de verbos HTTP:

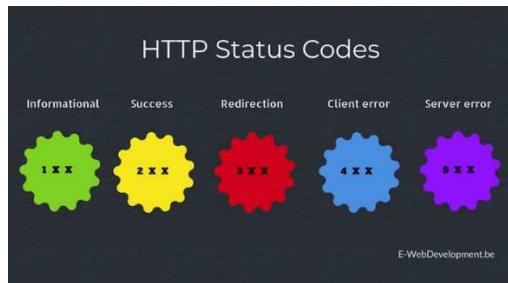
HTTP Verb	CRUD	Colección completa (/customers)	Recurso específico (/customers/{id})
POST	Create	201 (Created). Encabezado "Ubicación" con enlace a /customers/{id} que contiene el nuevo Id.	404 (Not Found), 409 (Conflict) si el recurso ya existe.
GET	Read	200 (OK). Lista de clientes. Utilice la paginación, la clasificación y el filtrado para navegar por listas grandes.	200 (OK), cliente único. 404 (Not Found), si el ID del cliente no existe o es inválido.

PUT	Update/Replace	405 (Method Not Allowed). A menos que desee actualizar o reemplazar todos los recursos de la colección completa.	200 (OK) or 204 (No Content). 404 (Not Found), si el ID del cliente no existe o es inválido.
PATCH	Update/Modify	405 (Method Not Allowed). A menos que desee modificar la colección en sí.	200 (OK) or 204 (No Content). 404 (Not Found), si el ID del cliente no existe o es inválido.
DELETE	Delete	405 (Method Not Allowed). A menos que desee eliminar toda la colección, lo que no suele ser deseable.	200 (OK). 404 (Not Found), si el ID del cliente no existe o es inválido.

5.6. Códigos de estatus HTTP.

Los códigos de estado de respuesta HTTP indican si se ha completado satisfactoriamente una solicitud HTTP específica. Las respuestas se agrupan en cinco clases:

- **Respuestas informativas (100–199).** Esta clase de código de estado indica una respuesta provisional, que consta solo de la línea de estado y encabezados opcionales, y termina con una línea vacía. Dicho de otra manera, son meramente informativas y nunca son usadas.
- **Respuestas satisfactorias (200–299).** Esta clase de código de estado indica que la solicitud del cliente fue recibida, comprendida y aceptada con éxito.
- **Redirecciones (300–399).** Esta clase de código de estado indica que el agente de usuario debe realizar más acciones para cumplir con la solicitud. La acción requerida PUEDE ser realizada por el agente de usuario sin interacción con el usuario si y solo si el método usado en la segunda solicitud es GET o HEAD. Un cliente DEBE detectar bucles de redirección infinitos, ya que dichos bucles generan tráfico de red para cada redirección.
- **Errores de clientes (400–499).** Esta clase de código de estado indica que la solicitud del cliente fue recibida, comprendida y aceptada pero que fue rechazada debido a un error de sintaxis incorrecta, o que la solicitud no puede cumplirse.
- **Errores de servidor (500–599).** Esta clase de código de estado indica que el agente de usuario realizó una solicitud pero tuvo un falló y que durante su solicitud, ocurrió un error interno durante su proceso de request.



En resumen:

HTTP Status Codes	
1xx - Informational	Significa que la solicitud ha sido recibida y el proceso continúa.
2xx - Success	Indica que la solicitud del cliente fue recibida, comprendida y aceptada con éxito
3xx - Redirection	Significa que se deben realizar más acciones para completar la solicitud.

4xx - Client error Solicitud no válida debido a un error de sintaxis incorrecta o esta solicitud no puede cumplirse.
5xx - Server error El servidor no ha cumplido una solicitud válida.

5.7. StateLess.

Cada mensaje que viaja a través de HTTP lleva toda la información necesaria para realizar la petición, es decir, el servidor no guarda datos referentes a otras comunicaciones con el cliente.

5.8. Consejos de diseño correcto de API REST.

El diseño de una API REST no es más que la creación de una interfaz con reglas bien definidas, puesta a disposición para que se pueda interactuar con un sistema y obtener su información o realizar operaciones. La API debe funcionar como un camarero en un restaurante: recibiendo sus pedidos y devolviendo los recursos (o platos), sin que los clientes tengan que visitar la cocina para recoger el plato allí.

Y, por supuesto, siempre es bueno tener en cuenta que, al diseñar una API REST, también debemos tener en cuenta quién la utilizará, cómo se utilizará dentro de la estrategia operacional y también cómo se expondrá al mundo. Este razonamiento, el de situar la API en un lugar privilegiado en la planificación, se denomina **API First (API Cliente)**.

Estos consejos son:

1. **Usar sustantivos y no verbos.** Uno de los principales fallos de estandarización a la hora de crear APIs RESTful está relacionado con el patrón de los *endpoints* creados (URLs de acceso al servicio). El estándar RESTful exige el uso de sustantivos, no de verbos o nombres de métodos.

Por ejemplo:

```
/getAllCustomers/createNewCustomer/eliminarCustomer
```

Esta es una forma incorrecta de nomenclatura, que se asemejan a las funciones de algún lenguaje de programación orientado a objetos. En su lugar, para el Diseño más apropiado, utilice sustantivos, como:

```
/clientes/clientes/563
```

2. **Utilizar correctamente los métodos HTTP.** El principio fundamental de REST es separar su API en recursos lógicos. Estos recursos se manipulan mediante peticiones HTTP con métodos **GET**, **POST**, **PUT** y **DELETE**. Por ejemplo, cuando se hace una petición al recurso `/clientes/563` utilizando el método **GET**, se está solicitando que se recupere un cliente específico con el código 563. Del mismo modo, cuando realice la misma petición (es decir, en el endpoint `/clientes/536`) utilizando el método **DELETE**, estará realizando una exclusión específica del cliente, el código 563.
3. **Utilizar los nombres en plural.** En general, la elección depende del arquitecto de software. En particular, es mejor usar los nombres en plural, ya que indican un conjunto de características (como en el caso de los clientes, más arriba). Sin embargo, una cosa es cierta: no mezclemos puntos finales en singular y en plural. **La recomendación es que simplifiquemos y usemos todos los nombres en plural.**
4. **Utilizar sub-recursos para las relaciones.** Hay situaciones en las que un recurso está relacionado con otro. Esto es habitual cuando existe una jerarquía de objetos y recursos. Por ejemplo, si se tratara de una API que devolviera datos estadísticos sobre la geografía de un país, podría haber sub-recursos para los estados dentro del país y los municipios dentro de los estados. Cuando proceda, utilicemos sub-recursos en los puntos finales. Por ejemplo, la solicitud:

```
GET /clientes/231/proyectos/
```

debe volver a la lista de proyectos del cliente 231. Y la solicitud

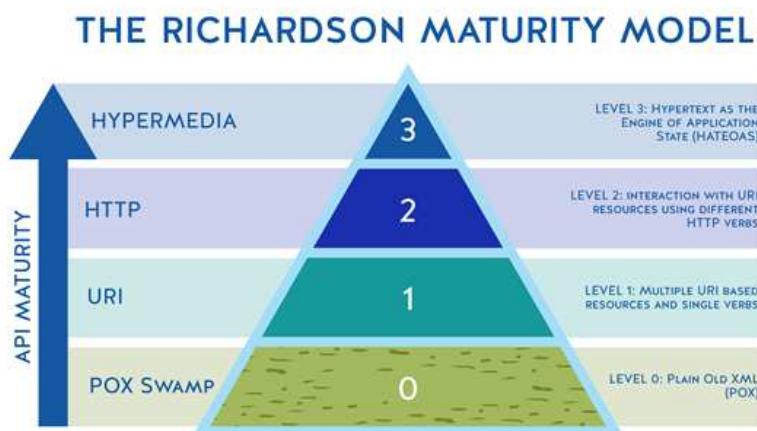
```
GET /clientes/231/proyectos/4
```

debe volver al proyecto nº 4 del cliente 231.

5. **No cambiar de estado con el método GET.** Cuando se realizan operaciones que cambian el estado de un objeto, se utilizan los métodos **POST**, **PUT** y **DELETE**. El método **GET**, como se intuye por su nombre, debe devolver sólo una versión de lectura del recurso. HTTP ofrece otros métodos para escribir en los recursos, así que utilízalos adecuadamente. En este punto, es importante recordar los permisos y cuestiones de seguridad de la API, lo que nos lleva a nuestro siguiente punto.
6. **Utilizar el cifrado SSL.** Una API RESTful debe utilizar necesariamente el cifrado SSL. Dado que se puede acceder a la web de su API desde cualquier lugar con acceso a Internet, como el patio de comidas de un centro comercial, una librería, una cafetería o un aeropuerto, su preocupación es garantizar un acceso seguro a los datos y servicios que ofrece su API. Sin embargo, no todos estos lugares ofrecen un acceso seguro y cifrado. Tener la información que se lleva encriptada **es esencial**. Además, el uso de SSL con tokens facilita la autenticación entre solicitudes, evitando que cada una de ellas tenga que volver a autenticarse.
7. **Crear versiones para la API.** Como todo software, las APIs deben crecer y evolucionar. Por muy cuidadoso y experimentado que sea, su API no será perfecta en la primera versión. A menudo, es mejor exponer la primera versión de la API y hacerla evolucionar gradualmente. Sin embargo, tengamos cuidado de no cambiar demasiado su diseño y acabar rompiendo las aplicaciones que utilizaban las versiones anteriores. Por lo tanto, al crear una nueva versión para la API, asegúrese de que la versión anterior sigue estando disponible, para no romper la funcionalidad del sistema. Tras comunicar los cambios a los desarrolladores, y darles tiempo para que se adapten, las versiones antiguas pueden ser descatalogadas, o pueden mantenerse en el aire, sin ofrecerles soporte..

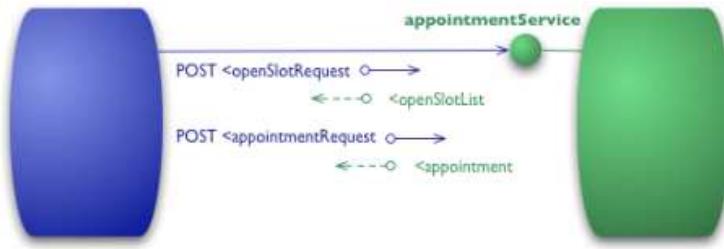
5.9. Modelo de madurez de Richardson.

Teniendo los conceptos anteriores ya recopilados, viene esto siguiente que es **MUY IMPORTANTE**: si bien la mayoría de los desarrolladores conocen REST, es posible que menos conozcan el **modelo de madurez de Richardson (Richardson Maturity Model)**. Aunque el modelo de madurez de Leonard Richardson a menudo se considera más esotérico, se puede entender mejor el concepto REST y por tanto conseguir llevar a cabo una implementación de los servicios mejor.

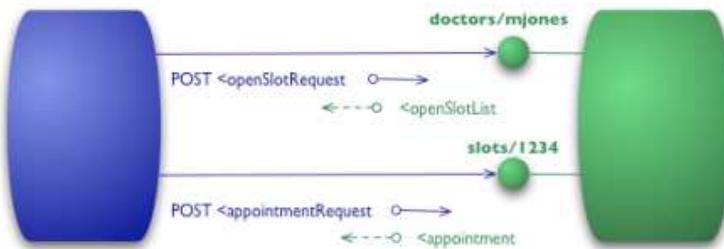


Este servicio se compone de cuatro niveles:

- **Nivel 0. El pantano de POX.** El modelo de madurez de Richardson no comienza en el nivel 1, sino en el nivel 0, el "pantano de la viruela". POX en este caso significa Plain Old XML y representa la funcionalidad más básica que se espera de una API que ingresa al Modelo de Madurez de Richardson. En este nivel, HTTP se utiliza como mecanismo de transporte para cada interacción remota, pero estas interacciones no utilizan los mecanismos inherentes a la web. HTTP sirve solo como un mecanismo básico de tunelización. En este nivel, toda la interacción de API es esencialmente RPC , una conversación XML de ida y vuelta en la que cada solicitud y respuesta no es más que un fragmento de código. Esto puede ser útil en algunos casos, seguramente, pero incluso en SOAP , la penúltima solución centrada en POX, cada iteración y evolución no es más que envolver el proceso HTTP en un sobre complicado.



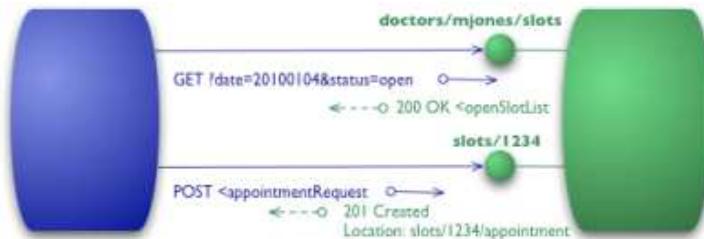
- **Nivel 1. Recursos.** Las conversaciones XML funcionan en algunos escenarios, pero no son ideales para las tareas complejas que se requieren en las interacciones y ofertas de API modernas. Por un lado, POX es ineficiente , a menudo transfiere texto superfluo. En segundo lugar, al comunicarse a través de XML, está complicando demasiado lo que debería ser una comunicación simple, eficiente y concisa. Nuestro primer paso para corregir estos problemas es salir del pantano de POX utilizando recursos . Los recursos permiten que las solicitudes se envíen al recurso específico en cuestión, en lugar de simplemente intercambiar datos entre cada punto final de servicio individual. Si bien esto parece una pequeña diferencia, en términos de función, es un cambio radical completo. En POX, una función mal definida simplemente se llama desde un cuerpo grande, un miasma similar al éter de recursos recopilados y posibles puntos de resolución, y luego esta función se define con más argumentos complejos. Al adoptar recursos en lugar de cuerpos XML para la comunicación, estamos estableciendo un tipo de identidad de objeto , llamando a un objeto en particular y pasándole argumentos que están específicamente relacionados solo con su función y forma.



- **Nivel 2. Verbos HTTP.** Ahora que nos hemos movido hacia los recursos, podemos comenzar a ver cómo interactuamos con esos recursos. En el enfoque de POX, la verborrea a menudo no importa específicamente. POST hace lo que POST hace, pero GET en muchos casos puede funcionar de la misma manera, lo que da como resultado una API que a menudo entrega los mismos datos para duplicar verbos. En el nivel 0 y el nivel 1, estos verbos sirven como un mecanismo de túnel para su solicitud; en esos niveles, esto está bien, ya que todo lo que la API realmente está haciendo es conversar en XML con XML. Sin embargo, el problema es que la verborrea es diferente por una razón. Cada verbo - GET, POST, PUT, DELETE, etc - hace algo muy específico, y estas funciones entregan a menudo totalmente diferentes conjuntos de resultados cuando se abordan adecuadamente - la falta de apalancamiento HTTP verbos por sus medios Usos correctos que usted está perdiendo una gran cantidad de funcionalidad potencial de ningún razón.

Como buen ejemplo, ambos POST y GET en el enfoque de POX darán como resultado el mismo retorno GET; sin embargo, se define como una operación segura, que abre un montón de funcionalidades. Una de esas funciones es el almacenamiento en caché , que se puede utilizar con gran efecto para reducir la cantidad de procesamiento que necesita la API, y un aumento en la eficiencia del tránsito al eliminar la necesidad de responder a solicitudes que solo generarán el mismo resultado.

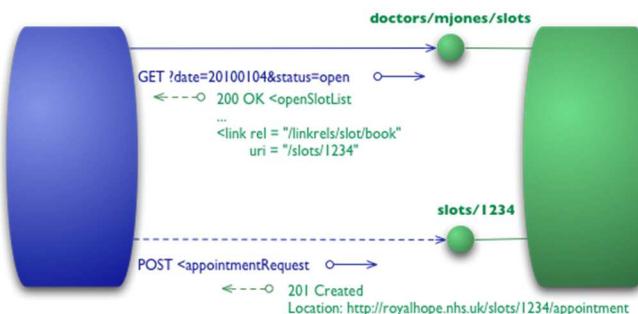
Todo eso se pierde, por supuesto, sin diseñar sus recursos específicamente en torno a la verborrea y la respuesta esperada. Al diferenciar el apalancamiento a través del diseño en el nivel 2, podemos aprovechar al máximo dichos beneficios.



- **Nivel 3. Hipermedia.** Para muchos desarrolladores, el nivel 2 es "suficientemente bueno". La verborrea adecuada y una reducción en la dependencia de la charla XML parece dar como resultado una API que está básicamente en un estado utilizable y, como tal, es donde muchos desarrolladores se sienten cómodos con detenerse. Desafortunadamente, esto no es todo el objetivo de este ejercicio: el paso final, el nivel 3, otorga ciertos beneficios que no deben ignorarse.

El nivel 3 implementa hipermedia , que puede usarse con gran efecto para extender la funcionalidad de una API a nuevos niveles. Al implementar HATEOAS (hipertexto como motor del estado de la aplicación), podemos hacer que la API responda a las solicitudes con información adicional y vincular recursos para obtener interacciones más ricas. Estos enlaces pueden hacer que la experiencia del usuario sea mucho más rica y pueden crear una red de funcionalidades que resulten en una más poderosa, más eficiente y más útil.

La utilización de hipermedia también ofrece la capacidad de actualizar esquemas de URI sin interrumpir a los clientes, lo que permite una mayor libertad para el cambio de desarrollo en el back-end. Este no es el caso antes del nivel 3, pero los beneficios no terminan ahí. El simple hecho es que una API que no utiliza Hypermedia está funcionando en su nivel más básico y menos útil: los fundamentos pueden estar disponibles, pero la API es autolimitante en extremo.



En resumen:

Nivel 0 – Swamp of POX

Se refiere a usar HTTP para interacciones remotas, pero sin usar otros mecanismos existentes para web.

Nivel 1 – Recursos.

La idea es identificar los recursos a través de un URI sin especificar la acción a realizar sobre el mismo.

Ejemplo. La URI

<http://www.misitio.com/personas/1>

representa a una persona y

<http://www.misitio.com/personas/2>

representa a otra persona en lugar de

<http://www.misitio.com/personas?id=1>

Nivel 2 – Verbos HTTP.

Este nivel nos indica que el API debería utilizar los verbos HTTP, mencionados anteriormente, utilizando correctamente los verbos y las respuestas.

Ejemplo:

- GET: <http://www.misitio.com/personas/1>
- DELETE: <http://www.misitio.com/personas/1>
- POST: <http://www.misitio.com/personas/1>
- PUT: <http://www.misitio.com/personas/1>

Evitando por ejemplo que si ocurre un error lo que se retorne sea un OK (200) que representa una solicitud correcta, sin errores.

Nivel 3 – Controles de hipermédia.

Básicamente es utilizar HATEOAS (Hipertexto como el mecanismo del estado de la aplicación).

Lo que indica HATEOAS es que al realizar un request, el mismo nos retorne información de como trabajar o manipular el recurso.

Ejemplo:

Si solicitamos datos de una persona <http://www.misitio.com/personas/1> a través de un Middleware intermedio, podemos determinar que permisos tiene la persona y que acciones puede realizar y así retornarlas. Aquí es donde entra HATEOAS para retornarnos acceso a diversas funcionalidades para el elemento, ya sea a través de HTML o XML/JSON.

`GET http://www.misitio.com/personas/1`

Resultado en JSON:

```
{  
  "nombre": "Walter",  
  "apellido": "Montes",  
  "href": "http://www.misitio.com/personas/1"}  
}
```

Con la referencia a la persona junto con la respuesta. Así cuando necesitemos realizar otra acción podemos utilizar los datos de URI del recurso.

Analizando nuestro proyecto, vemos que cumple correctamente este modelo y que nuestro deber es implementarlo de manera eficiente para que los servicios se implementen de la mejor manera. Este estandar es muy importante cuando hagamos proyectos de arquitectura limpia.

5.10. Regresando...

Con todo lo expuesto anteriormente, es hora de regresar a código fuente y aplicar estos conceptos antes mencionados a nuestra temática de este curso. Vemos lo siguiente:

- Hasta ahora, tenemos nuestros métodos y API controllers la petición GET, lo cual es bueno, puesto que muestra la información de las tiendas (Stores) y artículos (Articles). Y haciendo pruebas vemos que cumple hasta ahora los principios antes mencionados.
- Tenemos que meter un método POST en cada uno de los API Controllers respectivamente. Este método consiste en agregar un nuevo registro. En este caso, *agregar un nuevo artículo y una nueva sucursal*.

De este último punto, haremos lo siguiente:

1. En el proyecto **OA.Core**, carpeta **Interfaces**, agreguemos la siguiente línea de código en el módulo **IArticleRepository.cs**:

```
Task InsertArticle(Article article);
```

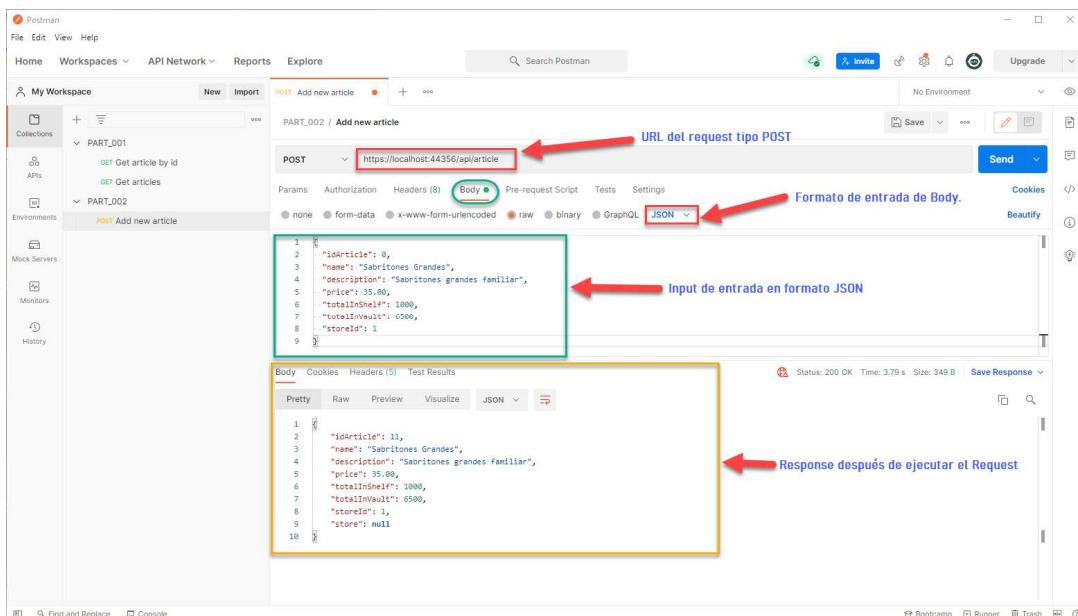
2. Implementar la lógica de la función en el módulo **ArticleRepository.cs** de **OA.Infrastructure\Repositories**:

```
public async Task InsertArticle(Article article)
{
    _context.Articles.Add(article);
    await _context.SaveChangesAsync();
}
```

3. En el módulo **ArticleController.cs** de **OA.Api\Controllers**, tenemos que crear un método **POST** para agregar un nuevo artículo y sería como esto:

```
[HttpPost]
public async Task<IActionResult> Post(Article article)
{
    await _articleRepository.InsertArticle(article);
    return Ok(article);
}
```

4. Guardemos los cambios y probemos esta nueva funcionalidad. Por medio de **Postman** ejecutemos el método **POST** para crear un nuevo artículo, creando un método **POST** como se muestra en la imagen. En el panel de **Body**, introducimos como objeto de entrada la estructura JSON del objeto Article de la siguiente manera:



5. Guardemos esa estructura como JSON. El request del endpoint que debe escribirse es <http://localhost:44356/article>. Guardemos ese request y lo ejecutamos. El input que metimos fue este:

```
{
```

```
        "idArticle": 0,
        "name": "Sabritones Grandes",
        "description": "Sabritones grandes familiar",
        "price": 35.00,
        "totalInShelf": 1000,
        "totalInVault": 6500,
        "storeId": 1
    }
```

Observemos lo siguiente:

- El método POST funciona correctamente. Eso es genial. Se guarda la información en nuestra Base de Datos.
- El detalle está que estamos usando como objeto de entrada un objeto de entidad, lo cual *es una mala practica*, puesto que estamos violando uno de los principios SOLID: el principio de responsabilidad, puesto que el objeto `Article` corresponde solamente a las entidades de dominio, y no a un objeto del tipo parámetro (*binding*).
- La ejecución del método POST funciona bien, pero tiene una seria vulnerabilidad: cualquier atacante puede meter información sin tener un control de validación de datos, y puede meter datos nulos, caracteres especiales, etc.

En el siguiente tópico, resolveremos estas cuestiones de seguridad antes mencionadas para prevenir este tipo de inconvenientes que comprometen la seguridad de la información de nuestra API RESTful del proyecto. Por el momento, guardemos los cambios y agreguemos un método POST para agregar nuevas sucursales, siguiendo los pasos mostrados anteriormente. Pero ademas... el usar un objeto del tipo Entidad como parámetro de entrada, viola el primer principio de SOLID puesto que la entidad solo se usa para representar un objeto de Base de Datos, no un parámetro de entrada para una petición del tipo POST, DELETE o PUT.

5.11. Resumen.

En este tópico, revisamos precisamente el punto medular de este tutorial: los conceptos de manera detallada de lo que es RESTful, API REST, y el principio de madurez de Richardson para mejorar y tener una buena calidad de servicio en nuestro proyecto de Clean Architecture. Es importante comentar que debemos tener esta base sólida sobre API REST puesto que nos ayudará como desarrolladores a crear mejores funciones API con mayor calidad y elegancia.

6. AutoMapper y DTO.

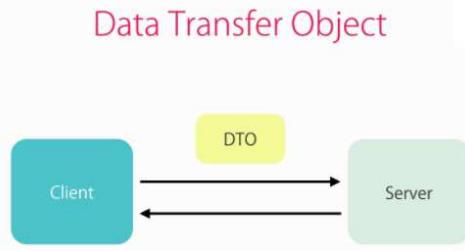
En esta sección, veremos un patrón de diseño muy importante cuando se trata del intercambio de datos entre una capa externa y una interna, que es llamado *Data Transfer Objects*, así como también como realizar una conversión de este tipo de objetos a los objetos de entidad de Base de Datos por medio de *Mapping*.

6.1. Concepto de DTO.

Un **objeto de transferencia de datos** (en inglés, **Data Transfer Object**, abreviado **DTO**) es un objeto que transporta datos entre procesos.

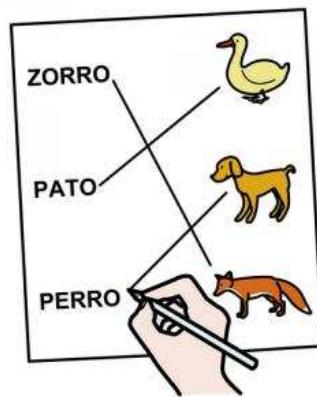
La motivación de su uso tiene relación con el hecho que la comunicación entre procesos se realiza generalmente mediante interfaces remotas (por ejemplo, servicios web), donde cada llamada es una operación costosa. Como la mayor parte del costo de cada llamada está relacionado con la comunicación de ida y vuelta entre el cliente y servidor, una forma de reducir el número de llamadas es usando un objeto (el **DTO**) que agrega los datos que habrían sido transferidos por cada llamada, pero que son entregados en una sola llamada.

La diferencia entre un objeto de transferencia de datos y un objeto de negocio (*Business Object*) o un objeto de acceso a datos (*Data Access Object*, DAO) es que un **DTO** no tiene más comportamiento que almacenar y entregar sus propios datos (métodos mutadores y accesores). Los **DTO** son objetos simples que no deben contener lógica de negocio que requiera pruebas generales.



6.2. Concepto de Mapping (Mapeo).

Revisemos estos conceptos importantes:



- **Mapeo (Mapping).** Significa tomar varias cosas y luego asociarlas de una manera u otra.
- **Mapeo de datos** es el proceso de extraer campos de datos de uno o varios archivos de origen y hacerlos coincidir con sus campos de destino. El mapeo de datos también ayuda a consolidar los datos extrayéndolos, transformándolos y cargándolos desde un sistema de origen a un sistema destino. El paso inicial de cualquier proceso de datos, es mapeo de datos. Las empresas pueden utilizar los datos mapeados para producir información relevante para mejorar la eficiencia empresarial.
- **Mapeo de información** es el trazado de un mapa de elementos de datos entre dos modelos de datos distintos.
- **Mapeo o asignación objeto-relacional**, es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia.



En la programación orientada a objetos, las tareas de gestión de datos son implementadas generalmente por la manipulación de objetos, los cuales son casi siempre valores no escalares. Para ilustrarlo, considere el ejemplo de una entrada en una libreta de direcciones, que representa a una sola persona con cero o más números telefónicos y cero o más direcciones. En una implementación orientada a objetos, esto puede ser modelado por un "objeto persona" con "campos" que almacenan los datos de dicha entrada: el nombre de la persona, una lista de números telefónicos y una lista de direcciones. La lista de números telefónicos estaría compuesta por "objetos de números telefónicos" y así sucesivamente. La entrada de la libreta de direcciones es tratada como un valor único por el lenguaje de programación (puede ser referenciada por una sola variable, por ejemplo). Se pueden asociar varios métodos al objeto, como uno que devuelva el número telefónico preferido, la dirección de su casa, etc.

Sin embargo, muchos productos populares de base de datos, como los Sistemas de Gestión de Bases de Datos SQL, solamente pueden almacenar y manipular valores escalares como enteros y cadenas, organizados en tablas normalizadas. El programador debe convertir los valores de los objetos en grupos de valores simples para almacenarlos en la base de datos (y volverlos a convertir luego de recuperarlos de la base de datos), o usar sólo valores escalares simples en el programa. El mapeo objeto-relacional es utilizado para implementar la primera aproximación.

El núcleo del problema reside en traducir estos objetos a formas que puedan ser almacenadas en la base de datos para recuperarlas fácilmente, mientras se preservan las propiedades de los objetos y sus relaciones; estos objetos se dice entonces que son persistentes.

6.3. Técnicas comunes de Mapeo de Datos.

Hay tres técnicas principales de mapeo de datos:

- **Asignación manual de datos.** Requiere que los profesionales de TI codifiquen o mapeen manualmente la fuente de datos al esquema de destino.
- **Mapeo de esquemas.** Es una estrategia semiautomatizada. Una solución de mapeo de datos establece una relación entre una fuente de datos y el esquema de destino. Los profesionales de TI comprueban las conexiones realizadas por la herramienta de mapeo de esquemas y realizan los ajustes necesarios.

- **Mapeo de datos completamente automatizado.** La técnica de mapeo de datos más conveniente, simple y eficiente utiliza una interfaz de usuario de mapeo de datos de arrastrar y soltar sin código. Incluso los usuarios no técnicos pueden realizar tareas de mapeo con solo unos pocos clics.

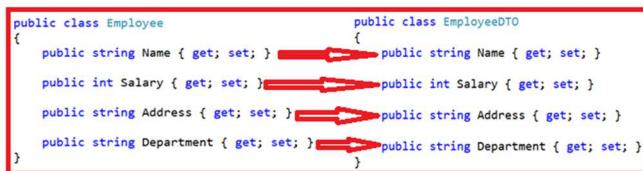
6.4. Casos de uso de Mapeo de Datos.

El mapeo permite a las empresas extraer valor comercial de los datos, ya que la información recopilada de diversas fuentes externas e internas debe unificarse y transformarse en un formato adecuado para los procesos operativos y analíticos. Estos son algunos de los casos de uso que utilizan ampliamente el proceso de mapeo:

- **Integración de Datos.** Para una integración exitosa, los repositorios de datos de origen y destino deben tener la misma estructura, lo cual es poco común. Herramientas de mapeo de datos ayudan a salvar las diferencias en los esquemas de los sistemas de origen y destino a través de la transformación y conversión de datos. Esto permite a las empresas consolidar la información de diferentes puntos de datos de manera eficiente. Es por eso que las herramientas de integración de datos disponibles en el mercado incluyen la función de mapeo sin código.
- **Migración de datos.** Migración de datos es el proceso de mover datos de una base de datos a otra, que se puede realizar sin problemas utilizando una herramienta de mapeo de bases de datos. Si bien hay varios pasos involucrados en el proceso, la creación de asignaciones entre el origen y el destino es una de las tareas más complejas y que requieren más tiempo, particularmente cuando se realiza manualmente. Los mapeos inexactos e inválidos en esta etapa pueden afectar negativamente la precisión y la integridad de los datos, lo que lleva al fracaso del proyecto de migración de datos. El software de mapeo de bases de datos sin código, con funciones de automatización, es una alternativa más segura para migrar datos con éxito a cualquier destino, como un almacén de datos.
- **Transformación de datos.** Dado que los datos empresariales residen en varias ubicaciones y formatos, el mapeo de datos y transformación de datos son esenciales para romper los silos de información y extraer conocimientos. El mapeo es el primer paso en el proceso de transformación de datos que lleva los datos a un área de preparación para convertirlos al formato deseado. Después de la transformación, se traslada al destino final, es decir, la base de datos.
- **Intercambio de datos electrónicos (EDI).** El mapeo de datos juega un papel importante en la conversión de archivos EDI al convertir los archivos a varios formatos, como XML, JSON y Excel. Una herramienta intuitiva de mapeo de datos permite al usuario extraer datos de diferentes fuentes y utilizar transformaciones y funciones integradas para mapear datos a Formatos EDI sin escribir una sola línea de código. Ayuda a realizar un intercambio de datos sin problemas.

6.5. AutoMapper.

Como su nombre lo dice **es un mapeador**. Es decir, permite para los valores seteados en una clase A a una clase B evitando la tediosa manera de hacerlo manualmente. Uno de los componentes más usados en .NET y .NET Core es precisamente AutoMapper, el cual, asigna las propiedades de dos objetos diferentes transformando el objeto de entrada de un tipo en el objeto de salida de otro tipo. También proporciona algunos datos interesantes para resolver el problema de cómo mapear un objeto de tipo A con un objeto de tipo B siempre que el objeto de tipo B siga la convención establecida de AutoMapper.



La manera de instalarlo en un proyecto de .NET Core es la siguiente:

```
$ dotnet add package AutoMapper
```

6.6. Nota importante.

A veces, al utilizar mapeadores, es muy difícil rastrear los errores, ya que no arrojan ninguna excepción, a menos que establezcamos explícitamente los mapeadores como mapeo estricto y durante ese tiempo, el código se compila y ejecuta correctamente, pero somos incapaces de encontrar esos errores. Esta situación ocurre rara vez con el mapeo manual, por lo que el mapeo manual de objetos tiene algunas ventajas sobre el mapeo automático.

6.7. Ajustando a nuestro código fuente.

Vamos entonces a realizar esta técnica del uso de DTO y el Mapping de objetos a objetos relacionales usando el componente de AutoMapper, siguiendo los siguientes pasos:

1. Agreguemos la referencia de Nuget llamada [AutoMapper.Extensions.Microsoft.DependencyInjection](#) en los proyectos **CA.Infrastructure** y **CA.API**, el cual, permitirá que AutoMapper adopte el patrón de inyección de dependencias.
2. En la carpeta **CA.Core\DTO**, vamos a crear los DTO correspondientes a los catálogos de artículos, sucursales y tipos de artículo. Por ejemplo, para la entidad Article, crearemos un archivo llamado **ArticleDTO.cs**, y copiemos desde el archivo **Article.cs** de la carpeta **Entities**, las propiedades solamente. Esto debe quedar mas o menos así:

```
using System;
namespace CA.Core.DTO
{
    public class ArticleDTO
    {
        public int SkuId { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public int TotalInShelf { get; set; }
        public int TotalInVault { get; set; }
        public int ProducttypeId { get; set; }
        public int StoreId { get; set; }
        public int AccountId { get; set; }
        public DateTime Creationdate { get; set; }
        public DateTime? Updatedate { get; set; }
    }
}
```

- Notemos que se parece mucho a **Article.cs** que se encuentra en la carpeta Entities del mismo proyecto. Solo estamos tomando las propiedades de la entidad con el mismo tipo de dato al DTO correspondiente.
3. Hay que crear una carpeta llamada **Mappings** en el proyecto **CA.Infrastructure**. Creamos un archivo llamado **AutoMapperProfile.cs**, el cual su contenido debe ser como este:

```

using AutoMapper;
using CA.Core.DTO;
using CA.Core.Entities;

namespace CA.Infrastructure.Mappings
{
    public class AutoMapperProfile : Profile
    {
        public AutoMapperProfile()
        {
            /* Articulos. */
            CreateMap<Article, ArticleDTO>();
            CreateMap<ArticleDTO, Article>();

            /* Sucursales. */
            CreateMap<Store, StoreDTO>();
            CreateMap<StoreDTO, Store>();

            /* Tipos de articulo. */
            CreateMap<ProductType, ProductTypeDTO>();
            CreateMap<ProductTypeDTO, ProductType>();

            /* Usuarios (Pendiente...). */
        }
    }
}

```

En el constructor de la clase, crearemos la configuración de mapeo de objetos como sigue: para en el caso de la entidad relacionada con los artículos, hay que crear el mapeo del objeto DTO a la entidad y realizar su proceso inverso, algo así como esto:

```

/* Articulos. */
CreateMap<Article, ArticleDTO>();
CreateMap<ArticleDTO, Article>();

```

- Guardemos los cambios y compilémoslos. Ahora tenemos que ir a la carpeta de **CA.API\Controllers**, en la aplicación de Web API y en **ArticleController.cs**, tenemos que declarar un atributo interno que haga referencia a AutoMapper:

```
private readonly IMapper _mapper;
```

Inyectemos ese objeto desde el constructor de la clase:

```

public ArticleController(IMapper mapper, IArticleRepository articleRepository)
{
    _mapper = mapper; _articleRepository = articleRepository;
}

```

Con esto ya podemos entonces realizar el mapeo de objetos de datos a entidad. Para el caso de los métodos GET hacemos esto:

```

[HttpGet]
public async Task<IActionResult> GetArticles()
{
    var _articles = await _articleRepository.GetArticlesAsync();
    var _articlesDTO = _mapper.Map<IEnumerable<ArticleDTO>>(_articles);
    return Ok(_articlesDTO);
}

[HttpGet("{id}")]
public async Task<IActionResult> GetArticle(int id)
{
    var _article = await _articleRepository.GetArticleAsync(id);
    var _articleDTO = _mapper.Map<ArticleDTO>(_article);
    return Ok(_articleDTO);
}

```

Y para el método POST:

```
[HttpPost]
public async Task<IActionResult> Post(ArticleDTO obj)
{
    var _article = _mapper.Map<Article>(obj);
    _article.Creationdate = DateTime.Now;
    await _articleRepository.AddArticle(_article);
    return Ok(obj);
}
```

Notemos que en los métodos GET, hicimos el mapeo de objetos desde la entidad al DTO, para mostrar los datos mapeados a ese DTO mismo con `_mapper.Map<T>`, para una lista de objetos del tipo `IEnumerable<T>` y un objeto T respectivamente. Para en el caso del POST, el procedimiento es inverso, puesto que ya no usaremos como parámetro de entrada (binding) el objeto de entidad `Article`, sino ahora será `ArticleDTO` y hacemos el mapeo de manera inversa, es decir, del DTO a la entidad de datos. Una vez hecho el mapeo correspondiente, se envía a la entidad de datos cargada a `_articleRepository.AddArticle` y se guardan los datos en Base de Datos. Nótese que una vez hecho el mapeo, asignamos el atributo `CreationDate` a la fecha actual y lo estamos aplicando al objeto de entidad, antes de guardarla a Base de Datos.

5. Guardemos los cambios. Hagamos los siguientes ajustes en el archivo `StartUp.cs`: en el método `ConfigureServices` agreguemos ahora la siguiente linea para realizar la inyección de dependencia para AutoMapper y debe ser así:

```
services.AddAutoMapper(typeof(Startup).Assembly, typeof(AutoMapperProfile).Assembly);
```

Esta opción es la más adecuada, puesto que estamos agregando las referencias de los tipos en el ensamblado actual: el primero es `StartUp`, haciendo referencia al módulo mismo de y al objeto `AutoMapperProfile`, que se encuentra en el ensamblado de `CA.Infrastructure`. Si deseamos que todos los objetos del tipo `Profile` de AutoMapper se incluyan en el arranque del proyecto, nuestra línea de código puede ser así:

```
services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
```

6. Guardemos los cambios de nuevo. Notemos que cuando compilamos y corremos el proyecto, vemos que el método GET que muestra la información de los artículos de la siguiente manera:

```
1 [
2   {
3     "skuId": 1,
4     "name": "Bolsa de churros Neto",
5     "description": "Bolsa de churros 100 grs",
6     "price": 25.5000,
7     "totalInShelf": 1000,
8     "totalInVault": 5000,
9     "productivityId": 1,
10    "storeId": 1,
11    "accountId": 1,
12    "creationdate": "2021-11-11T18:55:04.7",
13    "updatedate": null
14  },
15  {
16    "skuId": 2,
17    "name": "Pezataco Morado Grande",
18    "description": "Pezataco picoso ultrahot",
19    "price": 45.5000,
20    "totalInShelf": 1000,
21    "totalInVault": 5000,
22    "productivityId": 1,
23    "storeId": 1,
24    "accountId": 1,
25    "creationdate": "2021-11-11T18:55:04.7",
26    "updatedate": null
27 }
```

Vemos que nuestro JSON tiene atributos en nulo y otros atributos con arreglos vacíos. No tiene sentido mostrar la información así por que estamos mostrando **referencias circulares (loop reference)**, es decir, atributos repetitivos en algunos casos y otros sin sentido y esto, no es buena práctica, ya no digamos de lado de Backend, sino tambien de FrontEnd, cuando usan JavaScript para leer un JSON. Para corregir esto, tenemos que agregar una referencia a `CA.API` llamada `Microsoft.AspNetCore.Mvc.NewtonsoftJson` ejecutando el siguiente comando:

```
$ dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson
```

Ahora, incluyamos en el método ConfigureServices la siguiente línea:

```
services.AddControllers()
    .AddNewtonsoftJson(options => {
        options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore;
        options.SerializerSettings.NullValueHandling = Newtonsoft.Json.NullValueHandling.Ignore;
    });
}
```

Con esto, hacemos que el método `AddNewtonsoftJson` ajuste las opciones de serialización de JSON para quitar atributos nulos y referencias circulares, y que mostremos un JSON limpio. Guardemos de nuevo los cambios y ejecutamos de nuevo y notemos que cuando hacemos una petición GET, ya no se muestran las referencias circulares en el objeto JSON devuelto. Podemos aumentarle más configuraciones

7. Probemos ahora el método POST para agregar un nuevo artículo introduciendo algo como esto

```
{
    "name": "Sabritones grandes",
    "description": "Sabritones grandes tamano familiar",
    "price": 30.00,
    "totalInShelf": 4500,
    "totalInVault": 2500,
    "producttypeId": 1,
    "storeId": 1,
    "accountId": 1
}
```

Ejecutamos y vemos que ya se guarda los datos de DTO en Base de Datos, si ejecutamos de nuevo el método GET correspondiente, vemos que ya se muestra el registro nuevo en el JSON resultante.

8. Finalmente guardemos los cambios antes mencionados. Repetir el mismo procedimiento para los demás objetos de entidad de Base de Datos, generando propiamente dicho, los objetos DTO correspondientes y aplicandolo a los controllers respectivos y probar resultados.

Ahora, ya podemos entonces usar DTO's de manera fácil, haciendo el mapping con AutoMapper. Sin embargo, tenemos que solventar el problema de las validaciones de los datos que se introducen a la Web API, lo cual, es tema de la siguiente sección. Por ahora, vemos que ya nunca más usaremos la entidad de Base de Datos llamada `Article` como parámetro de entrada (ni cualquier otra entidad relacional) y cumplimos entonces el principio de SOLID de responsabilidad única para este caso.

6.8. Resumen.

Usar DTO y mapeo de datos es una técnica fácil que nos ahorra mucho a la hora de pasar datos en una Web API, lo cual, hace que sea mantenible el código y fácil de implementar, si en un futuro, se desean hacer más cambios al momento de introducir datos.

7. Autofilters y Validación Fluida.

En esta sección, veremos un patrón de diseño muy importante cuando se trata de validar información antes de hacer el intercambio de datos entre una capa externa y una interna, que es llamado *Fluent Validation*, así como también como entender el proceso de *filtering de ASP.NET Core* cuando se realizan las peticiones (request) para su supervisión y correcta ejecución de una API por medio de un componente llamado *AutoFilter*.

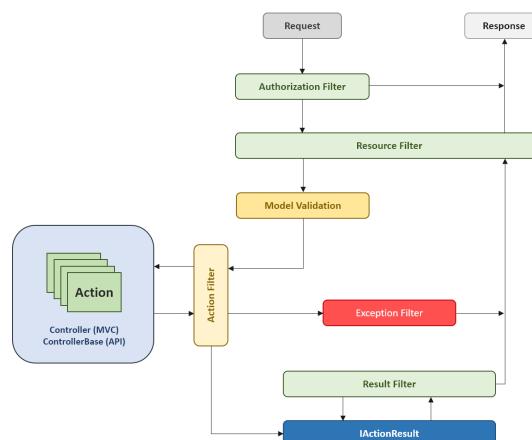
7.1. AutoFilters en ASP.NET Core.

Todo desarrollador backend de Web API en .NET Core debe comprender desde la generación del request hasta la devolución del response y que es lo que sucede entre esos dos momentos, es decir, como viene y hacia donde va y poder alterar el funcionamiento entre el envío de un request y la generación del response en un API. Todo esto se le conoce como **canalización de invocación de acciones de MVC y Web API**.

Los filtros en ASP.NET Core ofrecen una excelente manera de conectarse a la canalización de invocación de acciones de MVC y Web API. Por lo tanto, podemos usar filtros para extraer código que se pueda reutilizar y hacer que nuestras acciones sean más limpias y fáciles de mantener. Hay algunos filtros que ya son proporcionados por ASP.NET Core como el **filtro de autorización**, y están **los personalizados** que podemos crear nosotros mismos.

Existen diferentes tipos de filtros:

- **Filtros de autorización.** Se ejecutan primero para determinar si un usuario está autorizado para la solicitud actual.
- **Filtros de recursos.** Se ejecutan inmediatamente después de los filtros de autorización y son muy útiles para el almacenamiento en caché y el rendimiento.
- **Filtros de acción.** Se ejecutan justo antes y después de la ejecución del método de acción.
- **Filtros de excepción.** Se utilizan para manejar excepciones antes de que se complete el cuerpo de la respuesta.
- **Filtros de resultados.** Se ejecutan antes y después de la ejecución del resultado de los métodos de acción.



En este capítulo, hablaremos sobre los *filtros de acción* y cómo usarlos para crear código más limpio y reutilizable en nuestra API web.

7.2. Implementar un ActionFilter.

Para crear un ActionFilter, necesitamos crear una clase que herede de la interfaz `IActionFilter` o `IAsyncActionFilter` o de la clase `ActionFilterAttribute` que es la implementación de las dos interfaces antes mencionadas y de las siguientes interfaces:

```
public abstract class ActionFilterAttribute : Attribute,
    IActionFilter,
    IFilterMetadata,
    IAsyncActionFilter,
    IResultFilter,
    IAsyncResultFilter,
    IOrderedFilter
```

Para implementar un ActionFilter de manera sincrona, necesitamos implementar los métodos `OnActionExecuting` y `OnActionExecuted`. Esto indica que se ejecuta antes y después de la ejecución del método de acción:

```
namespace ActionFilters.Filters
{
    public class ActionFilterExample : IActionFilter
    {
        public void OnActionExecuting(ActionExecutingContext context)
        {
            // Nuestro código antes de que se ejecute la acción.
        }
        public void OnActionExecuted(ActionExecutedContext context)
        {
            // Nuestro código después de la acción se haya ejecutado.
        }
    }
}
```

Podemos hacer lo mismo con un ActionFilter asincrónico heredando de `IAsyncActionFilter`, pero solo tenemos un método para implementar `OnActionExecutionAsync`:

```
namespace ActionFilters.Filters
{
    public class AsyncActionFilterExample : IAsyncActionFilter
    {
        public async Task OnActionExecutionAsync(ActionExecutingContext context,
                                                ActionExecutionDelegate next)
        {
            // Ejecutar cualquier código antes de que se ejecute la acción.
            var result = await next();
            // Ejecutar cualquier código después de que se ejecute la acción
        }
    }
}
```

En nuestro caso, aplicando a nuestra problemática actual, usaremos un ActionFilter asincronico.

7.3. Ambito de los AutoFilters en .NET Core.

Al igual que los otros tipos de filtros, el filtro de acción se puede agregar a diferentes niveles de alcance:

- **Global.** Los filtros de acción se utilizan a nivel general de toda la WebAPI. Si queremos usar nuestro filtro globalmente, necesitamos registrarlo dentro del método `AddControllers()` en el método `ConfigureServices`, en `StartUp.cs`:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(config =>
    {
        config.Filters.Add(new GlobalFilterExample());
    });
}

```

- **Acción y Controlador.** Pero si queremos usar nuestro filtro como un tipo de servicio en el nivel de Acción o Controlador, necesitamos registrarlo en el mismo método `ConfigureServices` pero como un servicio en el contenedor de IoC:

```

services.AddScoped<ActionFilterExample>();
services.AddScoped<ControllerFilterExample>();

```

Finalmente, para usar un filtro registrado en el nivel de Acción o Controlador, debemos colocarlo encima del Controlador o Acción como `ServiceType`:

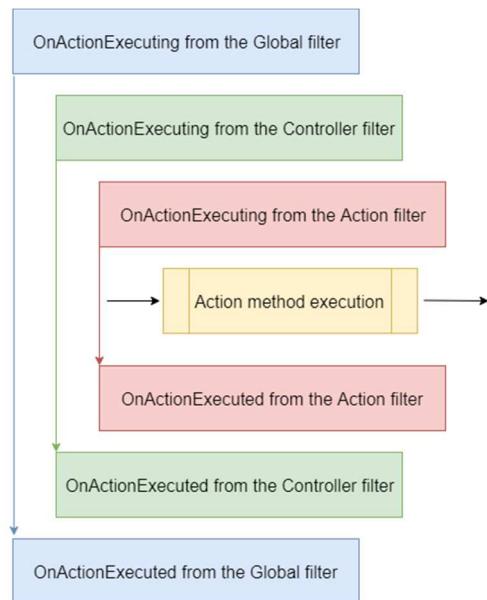
```

namespace AspNetCore.Controllers
{
    [ServiceFilter(typeof(ControllerFilterExample))]
    [Route("api/[controller]")]
    [ApiController]
    public class TestController : ControllerBase
    {
        [HttpGet]
        [ServiceFilter(typeof(ActionFilterExample))]
        public IEnumerable<string> Get()
        {
            return new string[] { "example", "data" };
        }
    }
}

```

7.4. Orden de invocación en ASP.NET Core.

El orden en el que se ejecutan nuestros AutoFilters es el siguiente:



Por supuesto, podemos cambiar el orden de invocación agregando una propiedad adicional `Order` a la declaración de invocación:

```

namespace AspNetCore.Controllers
{
    [ServiceFilter(typeof(ControllerFilterExample), Order=2)]
    [Route("api/[controller]")]
    [ApiController]
    public class TestController : ControllerBase
    {
        [HttpGet]
        [ServiceFilter(typeof(ActionFilterExample), Order=1)]
        public IEnumerable<string> Get()
        {
            return new string[] { "example", "data" };
        }
    }
}

```

O algo como esto además de la misma acción:

```

[HttpGet]
[ServiceFilter(typeof(ActionFilterExample), Order=2)]
[ServiceFilter(typeof(ActionFilterExample2), Order=1)]
public IEnumerable<string> Get()
{
    return new string[] { "example", "data" };
}

```

7.5. Fluent Validation.

El patrón de diseño de Validación Fluida (Fluent Validation Pattern) son conjuntos de técnicas y métodos que consisten en verificar que la integridad de los datos cumplan con ciertas condiciones, reglas o criterios antes de ser procesados. La validación de datos es esencial para cualquier aplicación. Cuando se trata de validar modelos, los desarrolladores suelen utilizar anotaciones de datos. Hay algunos problemas con el enfoque de anotaciones de datos:

- Las reglas de validación están estrechamente relacionadas con los modelos.
- Agregar complejidad a los modelos o objetos del tipo DTO.
- Difícil de realizar validaciones dinámicas y condicionales.
- Difícil de ampliar y escalar.

[FluentValidation](#) y [FluentValidation.AspNetCore](#) son componentes de .NET Core que reemplazan los atributos de validación existentes (DataAnnotations) y se basa en el patrón de diseño de API Fluida o Interfaz Fluida, así como de expresiones Lambda para crear reglas de validación fuertemente tipadas. La manera de incluirlos en un proyecto de .NET Core es la siguiente:

```
$ dotnet add package FluentValidation
$ dotnet add package FluentValidation.AspNetCore
```

Empezaremos a usar este componente para resolver el problema que teníamos en nuestro proyecto de Web API cuando los DTO's que creamos, no tenían un mecanismo de validación de datos antes de ejecutar las operaciones de Web API.

7.6. Ajustando nuestro código fuente: extendiendo el arranque del proyecto.

Vamos entonces a hacer un ajuste a nuestro proyecto de Web API: notamos que el archivo StartUp.cs, se esta extendiendo conforme introduzcamos más líneas de código en la función y esto tampoco no es una mejor práctica que digamos puesto que el código fuente se está haciendo mas largo, pesado y desordenado. La solución mas ideal sería crear una clase para cada característica o servicio agregado a `IServiceCollection`. Entonces vamos a hacerlo más modular como sigue:

1. En el proyecto **CA.Infrastructure**, hay que crear una carpeta llamada **Extensions** y dentro de ella crearemos dos carpetas más: **ApplicationBuilder** y **ServiceCollection**, respectivamente.

2. El archivo **IoC.cs** que se encontraba en **CA.API\Middleware** hay que pasarlo a **CA.Infrastructure\ServiceCollection**. Eliminemos la carpeta primera, **CA.API\Middleware** y en el archivo **IoC.cs** hay que cambiar, o mejor dicho, introducir estas líneas:

```
using CA.Core.Interfaces;
using CA.Infrastructure.Repositories;
```

Guardemos los cambios.

3. Revisemos en nuestro archivo **StartUp.cs**. Notamos que el método **ConfigureServices** se está extendiendo demasiado, en el aspecto de líneas de código fuente y es necesario pasar la configuración de los controllers y la configuración de los contextos de Base de Datos:

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    /* Añadimos "AutoMapper" antes de configurar los controllers y se cargan todas las configuraciones de
     * AutoMapper en las referencias de este ensamblado. */
    /* services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies()); */
    /* Otra forma */
    services.AddAutoMapper(typeof(Startup).Assembly, typeof(AutoMapperProfile).Assembly);

    /* Añadir controllers y configurando JSON para evitar referencias循ulares, ignorando atributos nulos y
     * formateando los atributos en notación "CamelCase", así como ajustando la zona horaria a meridiano local. */
    services.AddControllers()
        .AddNewtonsoftJson(options =>
    {
        options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore;
        options.SerializerSettings.NullValueHandling = Newtonsoft.Json.NullValueHandling.Ignore;
        options.SerializerSettings.DateTimeZoneHandling = Newtonsoft.Json.DateTimeZoneHandling.Local;
        options.UseCamelCasing(false);
    });

    /* Cadena de conexión al contexto de Base de Datos. */
    services.AddDbContext<PatosaDbContext>(options => {
        options.UseSqlServer(Configuration.GetConnectionString("PatosaDbContext"));
    });

    /* Contenedor de inversión de control (IoC) => Middleware. */
    IoC.AddDependency(services);
}
}
```

Entonces tenemos que crear dos archivos de clase más para esas dos secciones: **DbCtx.cs** para el contexto de Base de Datos y **CtrlCfg.cs** para almacenar la configuración de los controllers de nuestra Web API. Hay que crearlos en la carpeta **CA.Infrastructure\Extensions\ServiceCollection**.

4. Para crear **DbCtx.cs**, nuestro código debe ser así:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

using CA.Infrastructure.Data;

namespace CA.Infrastructure.Extensions.ServiceCollection
{
    public static class DbCtx
    {
        public static IServiceCollection AddDbContexts(this IServiceCollection services,
                                                       IConfiguration configuration)
        {
            services.AddDbContext<PatosaDbContext>(options => {
                options.UseSqlServer(configuration.GetConnectionString("PatosaDbContext"));
            });
            return services;
        }
    }
}
```

Lo que hicimos aquí fue lo siguiente: amplié **IServiceCollection** creando una clase estática llamada **DbCtx** e hice un método de extensión llamado **AddDbContexts**, pasando nuestro tipo (en este caso **IServiceCollection**) como parámetro con la palabra clave **this** y el otro parámetro llamado del tipo

`IConfiguration` para que leyera los valores del archivo de configuración de nuestra web API. Como esta función es del tipo `IServiceCollection`, debe devolverse del mismo tipo con la sentencia `return`.

5. Para crear `CtrlCfg.cs`, repitamos lo mismo y nuestro código debe quedar así:

```
using Microsoft.Extensions.DependencyInjection;
using CA.Infrastructure.Filters;

namespace CA.Infrastructure.Extensions.ServiceCollection
{
    public static class CtrlCfg
    {
        public static IServiceCollection AddControllersExtend(this IServiceCollection services)
        {
            services.AddControllers()
                .AddNewtonsoftJson(options =>
            {
                options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore;
                options.SerializerSettings.NullValueHandling = Newtonsoft.Json.NullValueHandling.Ignore;
                options.SerializerSettings.DateTimeZoneHandling = Newtonsoft.Json.DateTimeZoneHandling.Local;
                options.UseCamelCasing(false);
            });
            return services;
        }
    }
}
```

Aquí creamos el método del tipo estático de extensión llamada `AddControllersExtend`.

6. Guardemos los cambios y compilemos. Nuestro proyecto de la capa de Infraestructura debe los ajustes adecuados. Repitamos lo mismo ahora para la carpeta `ApplicationBuilder` y creamos un archivo llamado `DefaultCfg.cs`, con la siguiente estructura:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;

namespace CA.Infrastructure.Extensions.ApplicationBuilder
{
    public static class DefaultCfg
    {
        public static void InitConfigurationAPI(this IApplicationBuilder app,
                                                IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
                app.UseDeveloperExceptionPage();

            app.UseHttpsRedirection();
            app.UseRouting();
            app.UseAuthorization();
            app.UseEndpoints(endpoints => { endpoints.MapControllers(); });
        }
    }
}
```

Lo que estamos ahora haciendo es traer todo el código del método `Configure` de `StartUp.cs` y pasarlo a esta clase, haciendo lo mismo que en los pasos anteriores.

7. Guardemos los cambios y compilemos el proyecto. Antes de compilar, hay que asegurarnos que tengamos incluidas las dependencias de [Microsoft.AspNetCore.Mvc](#) y [Microsoft.AspNetCore.Mvc.NewtonsoftJson](#). No debería ocasionar errores después de incluirlas en el proyecto.
8. Regresemos a `StartUp.cs` y hagamos el siguiente ajuste en los métodos `ConfigureServices` y `Configure` respectivamente:

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

using CA.Infrastructure.Mappings;
using CA.Infrastructure.Extensions.ServiceCollection;
using CA.Infrastructure.Extensions.ApplicationBuilder;

namespace CA.Api
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            /* Añadimos "AutoMapper" antes de configurar los controllers y se cargan todas la configuraciones de
             * AutoMapper en las referencias de este ensamblado. */
            /* services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies()); */
            /* Otra forma */
            services.AddAutoMapper(typeof(Startup).Assembly, typeof(AutoMapperProfile).Assembly);

            /* Añadir controllers y configurando JSON para evitar referencias circulares, ignorando atributos nulos y
             * formateando los atributos en notación "CamelCase", así como ajustando la zona horaria a meridiano local. */
            CtrlCfg.AddControllersExtend(services);

            /* Cadena de conexión al contexto de Base de Datos. */
            DbCtx.AddDbContexts(services, Configuration);

            /* Contenedor de inversión de control (IoC) => Middleware. */
            IoC.AddDependency(services);
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            DefaultCfg.InitConfigurationAPI(app, env);
        }
    }
}

```

Notemos que se ve más limpio y con menos líneas. Al final quitaremos los comentarios pero vemos que con este ajuste, **StartUp.cs** ya se vé mas pulido. Como lo indica en el cuadro rojo, no olvidemos hacer referencia a los espacios de nombres de **CA.Infrastructure** para que no tengamos errores.

9. Ahora si: quitemos dependencias de nuestro proyecto **CA.API** y ajustemos los **using** en el archivo antes mencionado y compilemos todo. Con esto tenemos que nuestro proyecto de la capa mas externa listo y que de ahora en adelante podemos ya extender estos dos métodos de **StartUp.cs** en secciones más modulares.

Con esto, cumplimos el segundo principio de SOLID y tenemos un código mantenable y facil de seguir.

7.7. Creando nuestro ActionFilter global.

Ahora si, vamos a implementar nuestro ActionFilter como sigue: todo los controllers de nuestro proyecto de Web API tienen el decorador **[ApiController]** el cual indica que se puede aplicar a una clase de controlador para habilitar los siguientes comportamientos obstinados específicos de la API:

- Requisito de enrutamiento de atributos
- Respuestas HTTP 400 automáticas.
- Inferencia de parámetros de origen de enlace.
- Inferencia de solicitud de datos de formulario.

Normalmente, los archivos de clase de tipo **Controller** se derivan de la clase derivada **ControllerBase**. La diferencia entre esta clase derivada y **Controller** es que el primero es una base para un controlador MVC sin soporte de visualización. Cuando se tiene estos atributos de la primera clase derivada mencionada en una clase del tipo **Controller**, ya tenemos todas las funcionalidades de una API Controller que podemos trabajar dentro de nuestro archivo de clase.

Teniendo esto en claro, entonces vamos a construir nuestro ActionFilter global, en la cual, consiste en validar que todo modelo que ingrese a cualquiera de los controladores de nuestra Web API sean verificados como válidos. Hagamos esto:

1. En el proyecto **CA.Infrastructure**, creamos una carpeta llamada **Filters** y creamos un archivo llamado **GlobalValidationFilterAttribute.cs**, el cual su contenido es el siguiente:

```
using System.Linq;
using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace CA.Infrastructure.Filters
{
    public class GlobalValidationFilterAttribute : IAsyncActionFilter
    {
        public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
        {
            if (context.ActionArguments.Any(x => x.Value == null))
                context.Result = new BadRequestObjectResult("Object is null");

            if (!context.ModelState.IsValid)
            {
                context.Result = new BadRequestObjectResult(context.ModelState);
                return;
            }

            await next();
        }
    }
}
```

Notemos lo siguiente:

```
if (!context.ModelState.IsValid)
{
    context.Result = new BadRequestObjectResult(context.ModelState);
    return;
}
```

Estas líneas indican que si un **ModelState** no es válido, lanza una excepción del tipo **BadRequestObjectResult**. Las siguientes líneas:

```
if (context.ActionArguments.Any(x => x.Value == null))
    context.Result = new BadRequestObjectResult("Object is null");
```

Indican que si un objeto del contexto de entrada, es decir, los argumentos de entrada del contexto de la Web API son nulos, tambien devuelve una excepción del tipo **BadRequestObjectResult**. Esto lo podemos ampliarlo más adelante, para las operaciones CRUD, cuando ya introduzcamos contextos del tipo entidad.

2. Guardemos los cambios y para incluir ahora este ActionFilter, hay que registrarlo en el archivo **CtrlCfg.cs** dentro del método **AddControllers()**, de la siguiente manera:

```
using Microsoft.Extensions.DependencyInjection;
using CA.Infrastructure.Filters;

namespace CA.Infrastructure.Extensions.ServiceCollection
{
    public static class CtrlCfg
    {
        public static IServiceCollection AddControllersExtend(this IServiceCollection services)
        {
            services.AddControllers(options =>
            {
                options.Filters.Add<GlobalValidationFilterAttribute>();
            }).AddNewtonsoftJson(options =>
            {
                options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore;
                options.SerializerSettings.NullValueHandling = Newtonsoft.Json.NullValueHandling.Ignore;
                options.SerializerSettings.DateTimeZoneHandling = Newtonsoft.Json.DateTimeZoneHandling.Local;
                options.UseCamelCasing(false);
            });
            return services;
        }
    }
}
```

Guardemos cambios. Antes de probar este filtro de validación, tenemos que suprimir la validación del atributo `[ApiController]`. Si no lo hacemos, superará la validación de nuestro filtro de acción y siempre devolverá 400 (BadRequest) para todos los errores de validación. El modo de hacerlo es escribiendo algo como esto:

```
using Microsoft.Extensions.DependencyInjection;
using CA.Infrastructure.Filters;

namespace CA.Infrastructure.Extensions.ServiceCollection
{
    public static class CtrlCfg
    {
        public static IServiceCollection AddControllersExtend(this IServiceCollection services)
        {
            services.AddControllers(options =>
            {
                options.Filters.Add<GlobalValidationFilterAttribute>();
            }).ConfigureApiBehaviorOptions(options =>
            {
                options.SuppressModelStateInvalidFilter = true;
            }).AddNewtonsoftJson(options =>
            {
                options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore;
                options.SerializerSettings.NullValueHandling = Newtonsoft.Json.NullValueHandling.Ignore;
                options.SerializerSettings.DateTimeZoneHandling = Newtonsoft.Json.DateTimeZoneHandling.Local;
                options.UseCamelCasing(false);
            });
            return services;
        }
    }
}
```

Aquí lo que hicimos es agregar el método `ConfigureApiBehaviorOptions` y activando la opción `SuppressModelStateInvalidFilter` a `true`. Guardemos cambios y validemos los métodos POST que tenemos hasta ahora.

Con esto tenemos una primera barrera que valide nuestro métodos POST, DELETE o PUT cuando vayamos a realizar las operaciones CRUD en nuestra Web API. Sin embargo, si hacemos pruebas y metemos valores nulos, o quitamos atributos de los objetos DTO, volveremos a tener un error, ya no de validación por DataAnnotations, sino en Base de Datos, puesto que nuestra Base de Datos no acepta valores nulos en algunos campos y esto es un hueco seguridad todavía que tenemos que resolver. Para esto, usaremos el componente de `FluentValidator`. Más adelante, extenderemos mas la funcionalidad de nuestro ActionFilter cuando validemos si los objetos de entrada son del tipo Entidad y que no deban ser nulos, entre otras cosas.

7.8. Usando FluentAPIValidation.

El componente de FluentValidation para ASP.NET Core es fundamental para realizar esta complicada tarea de validar datos, por lo que hay que incluirla en nuestro proyecto `CA.Infrastructure` y dentro del mismo proyecto haremos lo siguiente:

1. Agreguemos a nuestro proyecto las referencias [FluentValidator](#) y [FluentValidator.AspNetCore](#).
2. Creamos una carpeta llamada **Validators**.
3. Creamos un archivo llamado **ArticleValidator.cs** y la estructura inicial de este archivo es la siguiente:

```

using System;
using FluentValidation;
using CA.Core.DTO;

namespace CA.Infrastructure.Validators
{
    public class ArticleValidator : AbstractValidator<ArticleDTO>
    {
        public ArticleValidator()
        {
            /* Reglas de cada uno de los atributos del objeto "ArticleDTO". */
        }
    }
}

```

Al incluir la clase derivada `AbstractValidator<T>`, estamos indicando que se está heredando la clase `AbstractValidator` para validar un objeto. En este caso `ArticleDTO`.

4. Hay que construir el constructor de la clase `ArticleValidator` y aquí ya podemos definir las reglas para cada atributo del objeto DTO. Para cada atributo, escribamos el siguiente código fuente:

```

RuleFor(u => u.SkuId) .Cascade(CascadeMode.Stop)
    .NotNull().When(u => u.UpdateDate != null).WithMessage("El identificador del articulo no puede ser nulo.")
    .GreaterThan(0).When(u => u.UpdateDate != null).WithMessage("El identificador del articulo no puede ser cero o negativo.");

```

Aquí lo que estamos creando son reglas, y se representan por la función `RuleFor`, con una expresión lambda donde se especifica la propiedad a validar y aquí ya podemos aplicar las validaciones sobre esta propiedad, ya sea predefinida o personalizada. No explicaré a detalle los métodos

5. El método `Cascade()` nos permite indicarle el comportamiento que queremos para nuestras reglas. Si no queremos que se muestre el resultado de la validación de todas las reglas, con sus errores establezcamos el valor a `CascadeMode.Stop`, el cual, mostrará solo el primer error de validación producido.
6. El método `NotNull()` indica que el atributo no debe aceptar valores nulos. También se puede usar `NotEmpty()` pero esto aplica para atributos de tipo de dato `String` (o texto).
7. El método `When()` indica que si este campo, en este caso `SkuId`, solo puede ser validado si el campo `UpdateDate` no es nulo. Recordemos que este último campo indica si tiene fecha de actualización de registro cuando se actualiza el registro. Lo que estamos haciendo aquí es si el campo no debe ser nulo, siempre y cuando la fecha de actualización no sea nula.
8. `GreaterThan()` indica que si el campo es mayor que cero y el campo `UpdateDate` no es nulo, no pasa la validación. Este método solo aplica a campos del tipo numérico y fecha en general.
9. Notemos que estos métodos usan `WithMessage()` para desplegar un mensaje de texto que indique que la validación falló.
10. En los atributos del tipo de dato `String`, la función `Length()` indica la longitud mínima y máxima que ese atributo debe tener como requisito obligatorio.
11. La función `Matches()` es una función que utiliza expresiones regulares para campos de texto, donde podemos validar si el campo de texto cumple con ciertas condiciones en base a los patrones de validación con expresiones regulares. Esta función es muy importante para asegurar que los datos que ingresen a nuestra Web API sean seguros.
12. Guardemos cambios y repitamos el procedimiento anterior para los DTO restantes.
13. Finalmente, regresemos al archivo `CtrlCfg.cs` para ya incluir la validación fluida por `FluentValidator`, con la siguiente función llamada `AddFluentValidation`:

```

using System.Reflection;
using FluentValidation.AspNetCore;
using Microsoft.Extensions.DependencyInjection;

using CA.Infrastructure.Filters;

namespace CA.Infrastructure.Extensions.ServiceCollection
{
    public static class CtrlCfg
    {
        public static IServiceCollection AddControllersExtend(this IServiceCollection services)
        {
            services.AddControllers(options =>
            {
                options.Filters.Add<GlobalValidationFilterAttribute>();
            }).ConfigureApiBehaviorOptions(options => {
                options.SuppressModelStateInvalidFilter = true;
            }).AddNewtonsoftJson(options =>
            {
                options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore;
                options.SerializerSettings.NullValueHandling = Newtonsoft.Json.NullValueHandling.Ignore;
                options.SerializerSettings.DateTimeZoneHandling = Newtonsoft.Json.DateTimeZoneHandling.Local;
                options.UseCamelCasing(false);
            }).AddFluentValidation(options => {
                options.RegisterValidatorsFromAssembly(Assembly.GetExecutingAssembly());
            });
            return services;
        }
    }
}

```

Lo cual indica que a nivel general, tomará todas las clases derivadas de **AbstractValidator** en todo el ensamblado.

14. Guardemos finalmente los cambios y compilemos. Si ejecutamos los métodos POST que tenemos hasta ahora registrados, notaremos que ahora si la validación fluida funciona correctamente y que ahora si, ya no tenemos problemas con los campos nulos, vacíos o bien atributos faltantes:

The screenshot shows a Postman request for 'PART_007 / Post Article'. The 'Body' tab is selected, showing a JSON payload:

```

1
2   "name": "Paquetaxos morados extra grandes",
3   "description": "<script>window.alert('MOLA')</script>",
4   "price": 30.00,
5   "totalInShelf": 4500,
6   "totalInVault": 2500,
7   "productTypeId": 1,
8   "storeId": 1,
9   "accountId": 1
10

```

The response tab shows a 400 Bad Request with the following validation error:

```

1
2   "Description": [
3       "Formato de descripción de artículo incorrecto."
4   ]
5

```

Con esto, ya tenemos la manera de validar la información de los DTO's que hemos construido y que cada vez que hagamos una petición POST, PUT o DELETE, se hace previamente un control de la validez del modelo DTO que se vaya a introducir en la Web API. Más adelante, haremos la separación de las operaciones de lectura y escritura para cada petición HTTP, una vez teniendo el CRUD construido.

7.9. Resumen.

El componente de FluentValidation para ASP.NET Core es fundamental para realizar esta complicada tarea de validar datos, pero tambien es importante entender que los filtros en ASP.NET Core son esenciales puesto que nos ayudan a ver como viaja la información desde un request hasta un response y que existen maneras de controlar el flujo de la ejecución de una llamada a una Web API por medio de estos componentes antes mencionados.

8. Persistencia de Datos: Unit of Work y Repository

En esta sección, veremos ahora si como vamos a construir la Persistencia de Datos en la capa de Infraestructura de nuestro proyecto Web API, así como consolidar el núcleo de las operaciones CRUD en nuestro proyecto para poder ya realizar esas operaciones y tener un control eficiente de las mismas.

8.1. Persistencia de Datos.

La **persistencia de datos** es un medio mediante el cual una aplicación puede recuperar información desde un sistema de almacenamiento no volátil y hacer que esta persista. La persistencia de datos es vital en las aplicaciones empresariales debido al acceso necesario a las bases de datos relacionales.

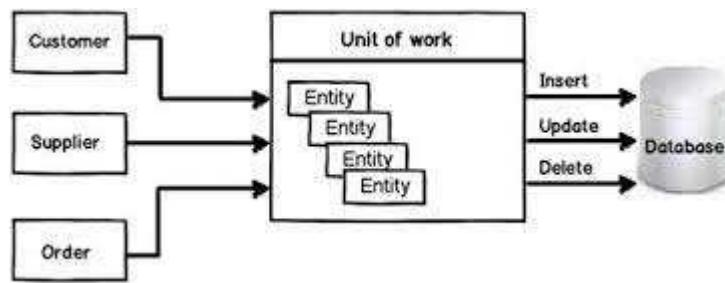
En Clean Architecture, es importante que esta sección se encuentre en la capa de Infraestructura, puesto que todo lo relacionado con Bases de Datos y sus entidades, se debe manejar de manera separada en esa capa. Entonces, esta parte debe estar compuesta de los siguientes elementos:

- Unit Of Work (UOW Pattern).
- Migraciones e instancia de la Base de Datos (Persistence).
- Repositorio por objeto de entidad en Base de Datos (Repository Pattern).

Estudiaremos a continuación cada uno de estos conceptos importantes.

8.2. Unit Of Work.

Cada método abre una instancia a la base de datos por lo cual podemos conllevar a problemas de performance. Más allá de esto, cada método gestiona su propia escritura de los cambios (problema real), es decir que al llamar a un método la acción para confirmar el INSERT/UPDATE/DELETE es ejecutado independientemente y esto no es muy bueno. Supongamos que quisieramos tener todo en un bloque de transaction por si algo falla, pues no, no vamos a poder hacerlo. Para resolver este problema, existe un patrón de diseño llamado **Unit Of Work (UOW)**.



UOW se puede definir como una sola transacción que involucra múltiples operaciones de inserción/actualización/eliminación, etc. Para decirlo en palabras simples, significa que para una acción de usuario específica (por ejemplo, el registro en un sitio web), todas las transacciones como insertar/actualizar/eliminar, etc., se realizan en una sola transacción, en lugar de realizar múltiples transacciones de base de datos. Esto significa que una unidad de trabajo aquí implica operaciones de inserción/actualización/eliminación, todo en una sola transacción.

En pocas palabras UOW te dice: “Yo voy a gestionar la instancia a la Base de Datos y hacerme cargo de las escrituras. Tú encárgate de la lógica del negocio”.

8.3. Patrones requeridos a usar.

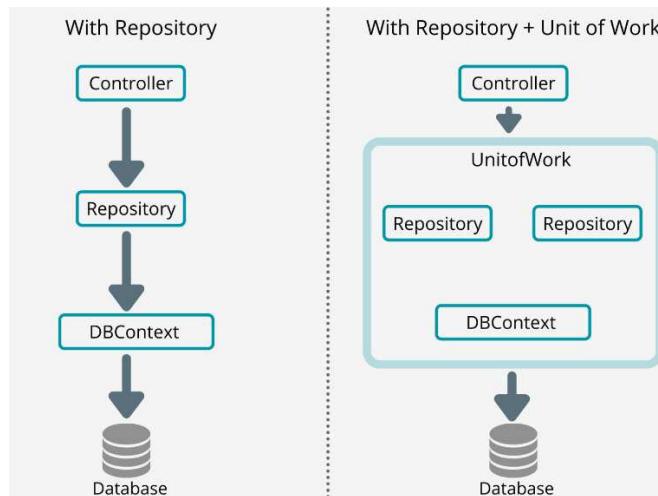
Los patrones de diseño siguientes para implementar UOW son los siguientes:

- **Services Pattern.** Nos permite organizar la lógica de nuestro negocio. Anteriormente, las consultas a la DB la armabamos en esta capa pero con este patrón de UnitOfWork lo que vamos a usar ahora son repositorios.
- **Repository Pattern.** Nos permite manipular el acceso a la base de datos; es decir, mediante este podemos realizar las consultas a la DB.
- **UnitOfWork Pattern.** Centraliza las conexiones a la base de datos y gestiona los cambios.
- **DbFactory Pattern.** Nos permite inicializar cada objeto de contexto de Base de Datos (DbContext) cuando lo usemos. Después, podemos deshacernos de él.

Del patrón Repository, existen dos tipos:

- **No genérico.** Este tipo de implementación implica el uso de una clase de repositorio para cada entidad. Por ejemplo, si tiene dos entidades, Empleado y Cliente, cada entidad tendrá su propio repositorio.
- **Générico.** Un repositorio genérico es aquel que se puede utilizar para todas las entidades. En otras palabras, puede usarse para Empleado o Cliente o cualquier otra entidad.

La siguiente imagen resume a groso modo la estructura de UOW:



8.4. Capas necesarias.

En la capa de Infraestructura del modelo de arquitectura de Clean Architecture, habíamos comentado que debe existir la persistencia de los datos. Hablando de este concepto, recordemos que la persistencia de datos es la capacidad de guardar información de un programa para poder usarse en otro momento y para un programador puede significar mas cosas y suele involucrar un proceso de **serialización de los datos** a un archivo o a una Base de Datos o un medio similar y el proceso inverso de recuperar los datos a partir de la información *serializada*. La persistencia de datos es en si la columna vertebral del modelo de Clean Architecture y en general, de cualquier arquitectura de la aplicación.

Bajo esta perspectiva, la persistencia de Datos se compone de los siguientes elementos, importantes:

- **Clients.** Es la sección donde se hace la implementación de nuestros clientes. En nuestro caso, es la aplicación de capa externa que es la Web API.
- **Services.** Capa donde se manipulan los repositorios. Aquí encapsulamos la lógica de llamadas a los repositorios mediante la capa de servicios (Service Layer), por que de esta manera, evitamos que llamen a los repositorios directamente desde el controller. Así mismo, podemos usar más de una lógica del repositorio desde un solo método de nuestra capa de servicios.
- **Persistencia de datos.** Consiste en los siguientes componentes, antes mencionados: UnitOfWork, las migraciones e instancia a Base de Datos y los repositorios, ya sean no genéricos o el repositorio genérico.

8.5. Sin embargo...

Todo lo que hemos mencionado suena interesante, sin embargo, mencionemos las ventajas y desventajas a continuación:

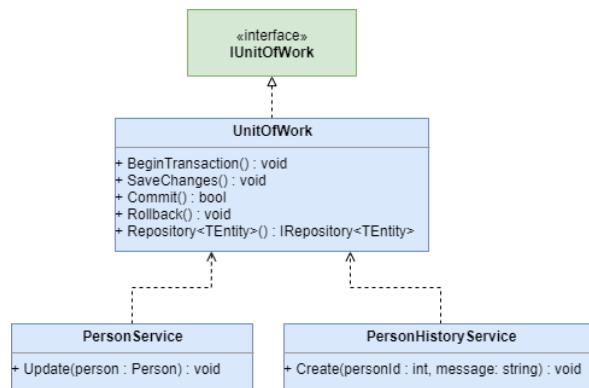
Ventajas:

- **La capa de Servicios encapsula toda la lógica de negocio,** de esta manera, evitamos tener el código acumulado directamente en nuestro controller o capa cliente.
- **El patrón UnitOfWork nos permite manipular mejor las conexiones a la Base de Datos** y realizar cambios cuando nosotros lo especificamos.
- **El uso del patrón de diseño Repository brinda la facilidad de reutilizar la lógica.** Por ejemplo, si quiero validar que un usuario existe en la Db y no tuvieramos esta capa tendría que duplicar el código por cada Service que lo vaya a utilizar. En cambio, al tenerlo este patrón implementado es más sencillo llamar desde cualquier servicio llamar a dicho método y si hacemos un cambio de este se actualizará para todos automáticamente (porque es una sola llamada). Eliminamos código repetitivo.
- Pruebas unitarias fáciles de realizar o TDD.
- Aumentar el nivel de abstracción y mantener la lógica de negocio por separado.

Desventajas:

- **Mayor inversión de código,** pero luego todo es reutilizable.
- **Algunos queries muy pesados pueden ser jodidos de implementar.** Por ejemplo, si quisiera un INNER JOIN de varias tablas para retornar un query con campos que no hacen referencia a una entidad, lo que estoy haciendo es crear una clase personalizada llamada por ejemplo UserReportAggregate y esta solo existiría desde el repositorio.
- Al quitar el acceso directo al DbContext vamos a tener que seguir optimizando nuestro repositorio genérico.

La arquitectura de UOW se centra en los cambios en los objetos. Como dijo Martin Fowler, esta unidad mantiene una lista de objetos modificados en un contexto de transacción. También gestiona la operación de escritura y se ocupa de los problemas de concurrencia. Podemos ver esto como un contexto, sesión u objeto que sigue todos los cambios en los modelos de datos durante una transacción.



8.6. DRY Principle.

El principio **No te repitas** (en inglés *Don't Repeat Yourself* o **DRY**) y conocido también como **Una vez y sólo una** es una filosofía de definición de procesos que promueve la reducción de la duplicación de código fuente en Desarrollo de Software. Dicho de otra manera, toda “pieza de información” nunca debería de ser duplicada o repetida debido a que incrementa la dificultad en los cambios y su evolución posterior y puede perjudicar la claridad del código fuente dando origen a inconsistencias y ambigüedades. Los términos “pieza de información” son usados en un sentido amplio, abarcando:

- Datos almacenados en una base de datos.
- Código fuente de un programa de software.
- Información textual o documentación.

Cuando el principio DRY se aplica de forma eficiente, los cambios en cualquier parte del proceso o código fuente requieren cambios en un único lugar. Por ejemplo, si algunas partes del proceso están por varias partes del código fuente, los cambios pueden provocar fallos con mayor facilidad si todos los sitios en los que aparece, no se encuentran sincronizados. Expliquemos esto con un ejemplo sencillo:

```
public class Product
{
    /* Unos atributos */
    public string Name { get; set; }

    public override string ToString()
    {
        return Name;
    }
}

public class Customer
{
    /* Otros atributos */
    public string Name { get; set; }

    public override string ToString()
    {
        return Name;
    }
}
```

Si nos damos cuenta las clases `Product` y `Customer` tienen dos miembros en común. Podríamos refactorizarlos y extraer esos miembros, así:

```
public class NamedEntity
{
    public string Name { get; set; }

    public override string ToString()
    {
        return Name;
    }
}

public class Product : NamedEntity
{
    /* Other atributos */
}

public class Customer : NamedEntity
{
    /* Other atributos */
}
```

Esta versión ahora cumple con el principio DRY. ¿Puede ser que dos clases tengan partes idénticas pero aún no se rompan este principio? Si. El principio DRY restringe la presencia de conocimiento de dominio, no pone límites al código real que se requiere para expresar ese conocimiento. El hecho de que las dos entidades tengan la misma funcionalidad no significa que violen DRY. De hecho, tanto las clases Producto como Cliente tienen su propia semántica. Simplemente sucedió que lo hicieron usando el código idéntico en este caso particular. Desde el punto de vista del dominio, estas entidades se desarrollan por separado unas de otras. Ambos representan diferentes partes del dominio y, por lo tanto, no violan el principio DRY.

Veamos ahora como se usa DRY para métodos, funciones e inclusive hasta clases o interfaces con otro ejemplo:

```
static Func<A, R> Memoize<A, R>(thisFunc<A, R> function)
{
    var cache = new Dictionary<A, R>();
    return argument =>
    {
        R result;
        if (!cache.TryGetValue(argument, out result))
        {
            result = function(argument);
            cache[argument] = result;
        }
        return result;
    };
}
```

Este ejemplo fue tomado del [artículo](#) de **Eric Lippert**. Aquí vemos que si el principio DRY nos obliga a mantener todos los conocimientos del dominio en un solo lugar, ¿qué pasa con el código que no contiene ninguno? ¿La duplicación de dicho código constituiría una violación de DRY? No, porque, nuevamente, el principio DRY se trata del conocimiento que es esencial para su dominio. Los métodos de utilidad no contienen tal conocimiento. Sin embargo, no significa que debamos introducir duplicaciones innecesarias en todos los métodos auxiliares. Simplemente significa que si no podemos evitarlo, no es gran cosa. Aquí, tenemos un método de utilidad bastante simple para memorizar la salida de una función. El problema es que si queremos introducir un memorizador para una función con 2, 3 o más parámetros, necesitaríamos escribir métodos que casi se dupliquen entre sí.

No hay forma de evitar esta duplicación. Al mismo tiempo, estos métodos no contienen ningún conocimiento sobre el dominio de la aplicación, por lo que está perfectamente bien tener varias versiones similares de ellos.

En resumen:

- El principio DRY tiene que ver con el conocimiento del dominio.
- No confunda adherirse a DRY con deshacerse de la repetición de código.
- Hay casos en los que la duplicación de código está correctamente aplicado.

8.7. El componente EntityBase.

Para llevar a cabo las operaciones CRUD de Base de Datos, es necesario tener un patrón de diseño más: **EntityBase** (entidad base). La ventaja de usar este componente es que todas las entidades de Base de Datos estén en igualdad de condiciones. Mejor lo expliquemos con este ejemplo:

```

/* Interface para crear un registro. */
public interface IAddEntity<TKey>
{
    public TKey Id { get; set; }
    public bool IsSystemRow { get; set; }
    public int AccountIdCreationDate { get; set; }
    public DateTime CreationDate { get; set; }
}

/* Interface para actualizar un registro. */
public interface IUpdateEntity<TKey> : IAddEntity<TKey>
{
    public DateTime? UpdateDate { get; set; }
    public int? AccountIdUpdateDate { get; set; }
}

/* Interface para eliminar un registro. */
public interface IDeleteEntity<TKey> : IAddEntity<TKey>
{
    public bool IsDeleted { get; set; }
    public DateTime? DeleteDate { get; set; }
    public int? AccountIdDeleteDate { get; set; }
}

/* Interface final como Entity Base. */
public interface IEntityBase<TKey> : IAddEntity<TKey>, IUpdateEntity<TKey>, IDeleteEntity<TKey> { }

```

La interfaz `IAddEntity<TKey>` es una plantilla para crear modelos de vista. El parámetro `TKey` aplica para cualquier tipo de dato y hace referencia al **identificador del registro en la entidad base**. A partir de esta interfaz de inicio, puedo crear otras interfaces que estén asociadas a `IAddEntity<TKey>` con su propia responsabilidad: `IUpdateEntity<TKey>` para actualizar registros y `IDeleteEntity<TKey>` para eliminación lógica de un registro. Finalmente, reunimos todas las interfaces en una sola llamada `IEntityBase<TKey>` y esta interfaz reúne todas las responsabilidades de las operaciones CRUD en Base de Datos. Su implementación sería la siguiente:

```

/* Implementación. */
public abstract class EntityBase<TKey> : IEntityBase<TKey>
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public TKey Id { get; set; }
    public bool IsSystemRow { get; set; }
    public int AccountIdCreationDate { get; set; }
    public DateTime CreationDate { get; set; }
    public DateTime? UpdateDate { get; set; }
    public int? AccountIdUpdateDate { get; set; }
    public bool IsDeleted { get; set; }
    public DateTime? DeleteDate { get; set; }
    public int? AccountIdDeleteDate { get; set; }
}

```

Aquí en esta implementación, si usamos `DataAnnotations` para que el atributo `Id` se use como campo del tipo identidad (`Identity`) y creamos entonces la clase `EntityBase<TKey>` que herede de la interfaz `IEntityBase<TKey>` y ya hereda las propiedades o atributos de las demás interfaces antes mencionadas. Como resultado de esto, estamos cumpliendo el cuarto principio de SOLID y DRY Principle. Si ahora, aplicamos esto en un objeto del tipo entidad, vemos que ya no es necesario repetir las mismas propiedades comunes en todas ellas:

```

public partial class Article : EntityBase<int>
{
    public Article() { }

    /* Los otros atributos de la clase 'Article'. */
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
    public int TotalInVault { get; set; }
    public int DepartamentId { get; set; }
    public int ProducttypeId { get; set; }
    public int SupplierId { get; set; }
    public int BrandId { get; set; }
    public string ImageArticle { get; set; }
}

```

Vemos que todas las entidades de la capa de Dominio ahora heredarán desde `Entity<int>` (o bien, cualquier tipo de dato, según lo que se crea conveniente). Por recomendación de optimización para Base de Datos, se recomienda preferentemente el tipo de dato entero, ya sea largo `long` o entero normal `int`.

En conclusión, no se puede exagerar el poder de crear estas interfaces y clases abstractas. No solo le permiten minimizar una gran cantidad de código repetido al principio, sino que también sirven para clasificar sus entidades en agrupaciones significativas que pueden ayudarnos a reducir aún más el código en el futuro. Por ejemplo, si tengo un fragmento de código que actúa sobre una entidad de "publicación", utilizando la interfaz, en lugar de la clase de entidad real, ahora puedo aplicar ese código ampliamente a cualquier otra entidad que también tenga propiedades de "publicación".

Según **Vladimir Khorikov**, en su libro *Unit Testing* (2014), introducir una clase base simplemente porque dos o más entidades tienen algunos miembros en común es generalmente una mala práctica. Se le llamaría en este caso, herencia de utilidad (**utility inheritance**). Para otros desarrolladores como Martin Fowler, Loc Nguyen y entre otros, es necesario aplicar este principio para los objetos entidad, puesto que es mejor reutilizar propiedades y métodos comunes o inicializar valores predeterminados al comprometerse con la Base de Datos. En fin: la decisión del arquitecto de software en cualquier lenguaje de programación de usar objetos de entidades base en sus proyectos es a consideración personal. Para nuestra temática, usaremos precisamente esta regla, contrario a lo que Khorikov dice, para separar las operaciones CRUD en Base de Datos.

8.8. El principio de Fail-Fast.

Cuando la persistencia de datos se implemente en una aplicación Backend, es importante saber que todas las operaciones que interactúan con ella, fallen en algún momento en tiempo de ejecución. Para esto, debe aplicarse el principio de **Fail-Fast (Falla rápida)**. Este principio consiste en detener una operación actual tan pronto como ocurra un error inesperado. La aplicación de este principio generalmente proporciona soluciones más estables cuando falle el código fuente en tiempo de ejecución. Los beneficios de este principio son los siguientes:

- **Acortando el ciclo de retroalimentación.** El costo de corregir un error encontrado mientras el software está en desarrollo es un orden de magnitud menor que cuando está en producción. Es notable la rapidez con la que se revelan los errores cuando se sigue el principio de falla rápida. Incluso si la aplicación está en producción, es mejor que informe a los usuarios finales si algo salió mal.
- **La confianza en que el software funciona según lo previsto.** Es posible que haya oído hablar de algunos lenguajes funcionales fuertemente tipados. Su sistema de tipos es tan estricto que si logra compilar un programa escrito en dicho lenguaje, lo más probable es que funcione correctamente. Podemos decir lo mismo de un programa escrito con el principio de falla rápida en mente. Si dicho programa aún se está ejecutando, lo más probable es que haga su trabajo correctamente.
- **La protección del estado de persistencia.** Si permitimos que el software continúe funcionando después de que ocurre un error, puede entrar en un estado no válido y, lo que es más importante, guardar ese estado en la base de datos. Esto, a su vez, conduce a un problema mayor, la corrupción de datos, que no se puede resolver con solo reiniciar la aplicación.

A consecuencia de este principio, veremos el siguiente patrón de diseño, muy importante en nuestros proyectos de código fuente.

8.9. Guard Clause Pattern.

Una **cláusula de protección** (del inglés **Guard Clause**) es una técnica derivada del método Fail-Fast y su propósito es validar una condición y detener inmediatamente la ejecución del código fuente, si la condición no se cumple, arrojando un error significativo en lugar de dejar que el programa arroje un error más profundo y menos significativo.

Las cláusulas de protección simplifican el código al eliminar las condiciones de ramificación anidadas innutiles, al devolver errores significativos. Veamos esto con un ejemplo sencillo:

```

public User GetUser(string userName)
{
    if (userName == null) throw new NullArgumentException(nameof(userName));
    // Continuar si se cumple la condición...
}

```

En este ejemplo, vemos que el bloque `if` actúa como una cláusula de protección al proteger el método `GetUser` contra cualquier argumento de nombre de usuario nulo. Así, somos libres de escribir el resto del método sin tener que preocuparnos por verificar si `UserName` es nulo.

Como regla principal, **las excepciones de las cláusulas de protección, nunca deben detectarse**. Dicho de otra manera, la mayoría de las veces, debe dejar que se haga hincapié en esas excepciones porque la mayoría de las veces, las cláusulas de protección protegerán contra escenarios que nunca deberían suceder como argumentos nulos. ¿Qué significa un argumento nulo? La mayoría de las veces, un argumento nulo es un error, así que ¿deberíamos detectar un error y arriesgarnos a no descubrirlo nunca? ¡No! En cambio, queremos dejar que la aplicación falle inmediatamente para que podamos descubrir el error antes de implementarla en producción durante el proceso de desarrollo.

Pero, ¿qué pasa si tenemos condiciones previas que no dependen de los errores? ¿Qué pasa si tenemos condiciones previas que podrían ocurrir a veces, como las condiciones previas de la lógica empresarial? ¡La solución es exponer sus cláusulas de protección!

¿Por qué exponer las cláusulas de protección? A veces, debe protegerse contra la lógica empresarial, lo que significa que es posible que la condición no se respete y no sea necesariamente un error. En esos casos, una posibilidad es exponer las cláusulas de protección relacionadas bajo un método booleano público para permitir que quién llame a esa función, se bifurque en torno a este método booleano. Veamos este ejemplo de crear una cuenta de usuario y su implementación:

```

public bool CanCreateUser(string userName, string password)
{
    return !this.UserNameExists(userName) && this.IsStrongPassword(password);
}

public void CreateUser(string userName, string password)
{
    if (!this.CanCreateUser(userName, password))
        throw new Exception("La cuenta de usuario no puede crearse.");
    // Creamos el usuario...
}

```

En este caso, su uso sería así:

```

if (!CanCreateUser(userName, password))
{
    Console.WriteLine("Usuario o contraseña inválidos.");
    return;
}
CreateUser(userName, password);
Console.WriteLine("La cuenta de usuario se ha creado correctamente.");

```

¿De qué tenemos que protegernos? Guard Clause nos previene de las siguientes situaciones:

- **Condiciones previas.** Las condiciones previas son las condiciones que deben cumplirse antes de la ejecución del método. Básicamente, las condiciones previas del método siempre dependerán de los argumentos del método. Un buen ejemplo de una condición previa de método es un argumento no nulo.

```

public User GetUser(string userName)
{
    // Requiere un argumento 'userName' no nulo.
    if (userName == null) throw new NullArgumentException(nameof(userName));
    // Ejecutar el propósito del método...
}

```

- **Condiciones posteriores.** Las condiciones previas son las condiciones que deben cumplirse después de la ejecución del método. Básicamente, las condiciones posteriores al método siempre dependerán del valor devuelto por el método. Un buen ejemplo de una condición posterior a un método es una cadena no nula o no vacía.

```
public User GetUsers()
{
    // Ejecutar el propósito del método...
    var users = this._repository.GetUsers();
    // Nos aseguramos que 'UserRepository.GetUsers()' no devuelva usuarios nulos.
    if (users.Any(user => user == null))
        throw new Exception("UserRepository.GetResult() devolvió una colección de usuarios nulos.");
    return users;
}
```

Como podemos adivinar, las condiciones posteriores no son obligatorias, ya que sabe exactamente lo que sale del método. La única vez que debe considerar el uso de condiciones posteriores es cuando tiene que usar llamadas no confiables (es decir, métodos externos o métodos protegidos) para obtener el resultado del método como en el ejemplo anterior.

¿Por qué hay que crearlos? Es una buena práctica encapsular nuestras cláusulas de protección dentro de una clase dedicada a este patrón de diseño para que podamos reutilizar la lógica y escribirlas de manera más legible. Veamos este ejemplo:

```
public static Guard
{
    public static void Requires(Func<bool> predicate, string message)
    {
        if (predicate())
            return;
        Debug.Fail(message);
        throw new GuardClauseException(message);
    }

    [Conditional("DEBUG")]
    public static void Ensures(Func<bool> predicate, string message)
    {
        Debug.Assert(predicate(), message);
    }
}
```

En esa implementación, el método `Requires` se usa para validar las condiciones previas y el método `Ensures` es para validar las condiciones posteriores. El punto interesante de esta implementación es el uso de la clase C# `Debug` proveniente del espacio de nombres `System.Diagnostics`. El punto principal de la clase `Debug` es que se ejecutará solo en modo de depuración y no puede ser detectado por alguien que lo invoque, por lo que se respeta la regla de *never detectar excepciones de cláusulas de protección*. Además, el método `Ensures` usa el atributo `Conditional` para asegurarse de que el código dentro del método se ejecutará solo en la depuración para que el rendimiento no se vea afectado en producción porque recordemos que los errores de las cláusulas de protección se detectarán antes de implementar en producción. Pero solo para estar seguros, nos protegemos en el método `Requires` lanzando una excepción `GuardClauseException` si no se cumple la condición previa porque el código posterior probablemente fallará de todos modos o causará una inconsistencia de datos que no queremos. La implementación final sería así:

```
public User GetUserRange(int from, int to)
{
    Guard.Requires(() => from >= 0, "Recibí un valor negativo desde... ");
    Guard.Requires(() => to >= 0, "Recibí un valor negativo hasta... ");

    var users = this._repository.GetUsers();
    foreach (user in result)
        Guard.Ensures(
            () => user != null,
            "UserRepository.GetResult() no puede ser nulo.");

    return users;
}
```

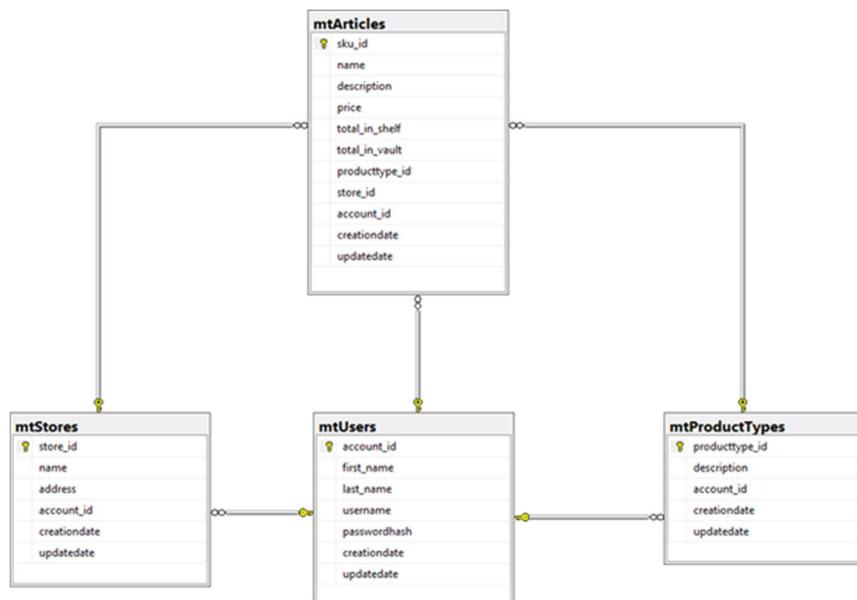
En resumen: las cláusulas de protección pueden ayudarnos a descubrir errores antes de implementarlo en producción y hacer que nuestro código sea más legible. También hemos aprendido por qué las excepciones de las cláusulas de protección nunca deben manejarse y cómo exponer las cláusulas de protección como un inconveniente cuando se basan en la lógica empresarial. En .NET Core, existe un componente para este patrón de diseño llamado [Ardalis.GuardClauses](#) que usaremos en nuestro proyecto backend Web API:

```
$ dotnet add package Ardalис.GuardClauses
```

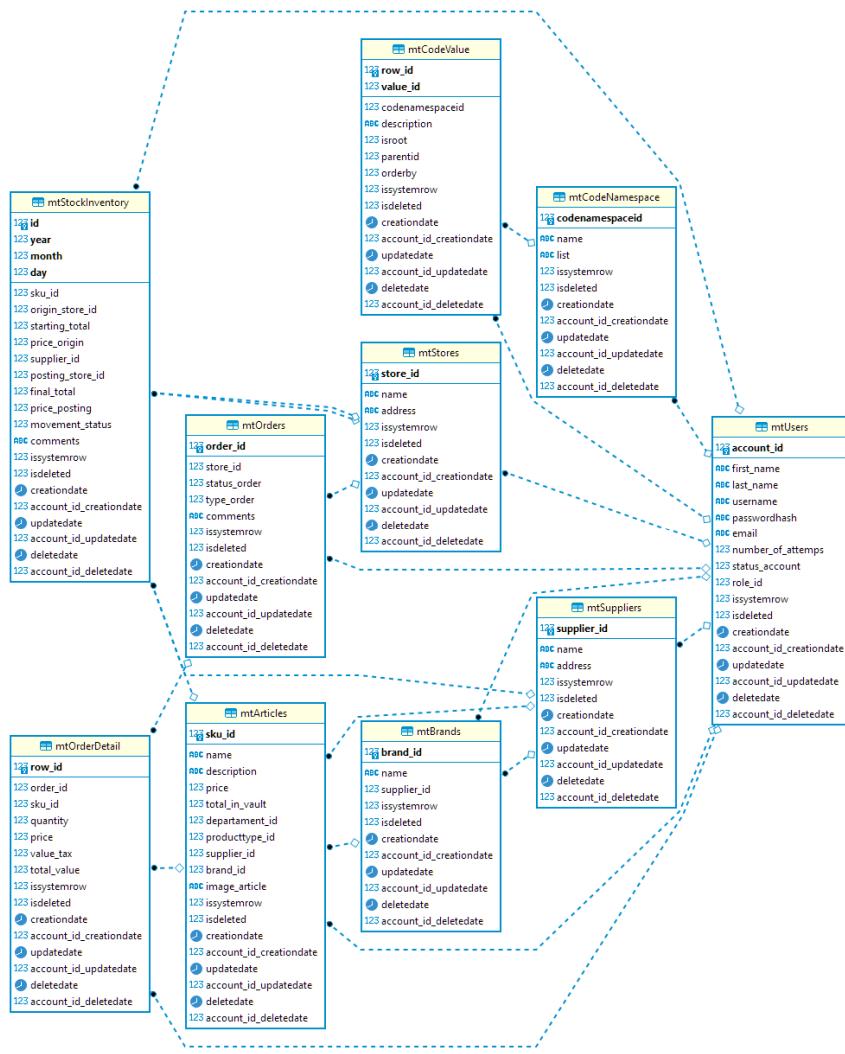
8.10. Antes de empezar...

Finalmente, llegamos ahora si a implementar lo que hemos visto en teoría. Pero antes, tenemos que hacer algunos ajustes críticos antes de construir nuestra Persistencia de Datos basada en Unit Of Work:

PASO 1. Refactorizar la base de Datos. Nuestra Base de Datos **PatosaComercial** hasta ahora nos ha servido para guardar la información solicitada. Sin embargo, ahora que vamos a robustecer la persistencia de datos, tenemos que refactorizar las tablas existentes y crear nuevas tablas. Esta era la estructura anterior:



La nueva estructura de datos es la siguiente:



Resumiendo, las tablas serían así:

- Catálogo de usuarios.

mtUsers (Catálogo de cuentas de usuario)	
Campo:	Descripción:
account_id	Identificador de la cuenta de usuario.
first_name	Nombre del usuario
last_name	Apellidos del usuario.
username	Cuenta de usuario.
passwordhash	Contraseña de la cuenta de usuario, cifrado
email	Correo electrónico de la cuenta de usuario.
number_of_attempts	Número de intentos de sesión al sistema.
status_account	Estatus de la cuenta de usuario actual.
role_id	Rol de la cuenta de usuario asignado.
isSystemRow	Flag que indica si es un registro crítico de la aplicación.
isDeleted	Flag que indica si es un registro eliminado lógicamente.
creationdate	Fecha en que se hizo la operación de ADD del registro.
account_id_creationdate	Identificador de la cuenta de usuario para el alta de nuevo registro.
updatedate	Fecha en que se hizo la operación de UPDATE del registro (nulo)
account_id_updatedate	Identificador de la cuenta de usuario para la actualización.
deletedate	Fecha en que se hizo la operación de DELETE lógico del registro (nulo)
account_id_deletedate	Identificador de la cuenta de usuario para la eliminación lógica.

- Catálogo de códigos de espacio de nombres.

mtCodeNamespace (Catálogo de códigos de espacio de nombres).	
Campo:	Descripción:
codenamespaceid	Identificador del espacio de nombres.
name	Nombre del campo de la tabla que hace referencia a la enumeración.
list	Nombre de la tabla en Base de Datos.
issystemrow	Flag que Indica si es un registro crítico de la aplicación.
isdeleted	Flag que Indica si es un registro eliminado lógicamente.
creationdate	Fecha en que se hizo la operación de ADD del registro.
account_id_creationdate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para el alta de nuevo registro.
updatedate	Fecha en que se hizo la operación de UPDATE del registro (nulo)
account_id_updatedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la actualización.
deletedate	Fecha en que se hizo la operación de DELETE lógico del registro (nulo)
account_id_deletedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la eliminación lógica.

- Catálogo de códigos de valor.

mtCodeValue (Catálogo de códigos de valor).	
Campo:	Descripción:
row_id	Identificador del registro.
value_id	Identificador de la opción.
codenamespaceid	Identificador del espacio de nombres.
description	Nombre de la opción.
isroot	Flag que indica si es un elemento padre.
parentid	Identificador de nodo raíz.
orderby	Número de orden de enumeración.
issystemrow	Flag que indica si es un registro crítico de la aplicación.
isdeleted	Flag que indica si es un registro eliminado lógicamente.
creationdate	Fecha en que se hizo la operación de ADD del registro.
account_id_creationdate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para el alta de nuevo registro.
updatedate	Fecha en que se hizo la operación de UPDATE del registro (nulo)
account_id_updatedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la actualización.
deletedate	Fecha en que se hizo la operación de DELETE lógico del registro (nulo)
account_id_deletedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la eliminación lógica.

- Catálogo de sucursales.

mtStores (Catálogo de sucursales).	
Campo:	Descripción:
store_id	Identificador de la sucursal.
name	Nombre de la sucursal.
address	Domicilio de la sucursal.
issystemrow	Flag que indica si es un registro crítico de la aplicación.
isdeleted	Flag que indica si es un registro eliminado lógicamente.
creationdate	Fecha en que se hizo la operación de ADD del registro.
account_id_creationdate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para el alta de nuevo registro.
updatedate	Fecha en que se hizo la operación de UPDATE del registro (nulo)
account_id_updatedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la actualización.
deletedate	Fecha en que se hizo la operación de DELETE lógico del registro (nulo)
account_id_deletedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la eliminación lógica.

- Catálogo de proveedores.

mtSuppliers (Catálogo de proveedores).	
Campo:	Descripción:
supplier_id	Identificador del proveedor.
name	Nombre del proveedor.
address	Domicilio del proveedor.
issystemrow	Flag que Indica si es un registro crítico de la aplicación.
isdeleted	Flag que Indica si es un registro eliminado lógicamente.
creationdate	Fecha en que se hizo la operación de ADD del registro.
account_id_creationdate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para el alta de nuevo registro.
updatedate	Fecha en que se hizo la operación de UPDATE del registro (nulo)
account_id_updatedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la actualización.
deletedate	Fecha en que se hizo la operación de DELETE lógico del registro (nulo)
account_id_deletedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la eliminación lógica.

- Catálogo de marcas de artículos.

mtBrands (Catálogo de marcas de artículos).	
Campo:	Descripción:
brand_id	Identificador de la marca de artículo.
name	Nombre de la marca de artículo.
supplier_id	Identificador del proveedor, asociado a la marca.
isSystemRow	Flag que indica si es un registro crítico de la aplicación.
isDeleted	Flag que indica si es un registro eliminado lógicamente.
creationDate	Fecha en que se hizo la operación de ADD del registro.
account_id_creationdate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para el alta de nuevo registro.
updatedate	Fecha en que se hizo la operación de UPDATE del registro (nulo)
account_id_updatedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la actualización.
deletedate	Fecha en que se hizo la operación de DELETE lógico del registro (nulo)
account_id_deletedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la eliminación lógica.

- Catálogo de artículos.

mtArticles (Catálogo de artículos).	
Campo:	Descripción:
sku_id	Identificador del artículo.
name	Nombre corto del artículo.
description	Nombre largo del artículo.
price	Precio unitario del artículo.
total_in_vault	Total de existencias en bodega, al momento de darse de alta como nuevo artículo.
departament_id	Identificador del departamento del artículo.
producttype_id	Identificador del tipo de producto del artículo.
supplier_id	Identificador del proveedor del artículo.
brand_id	Identificador de la marca del artículo.
image_article	Imagen del artículo, en base64 texto.
isSystemRow	Flag que indica si es un registro crítico de la aplicación.
isDeleted	Flag que indica si es un registro eliminado lógicamente.
creationDate	Fecha en que se hizo la operación de ADD del registro.
account_id_creationdate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para el alta de nuevo registro.
updatedate	Fecha en que se hizo la operación de UPDATE del registro (nulo)
account_id_updatedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la actualización.
deletedate	Fecha en que se hizo la operación de DELETE lógico del registro (nulo)
account_id_deletedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la eliminación lógica.

- Catálogo de movimientos de stock.

mtStockInventory (Catálogo de movimientos de almacen).	
Campo:	Descripción:
id	Identificador del movimiento.
sku_id	Identificador del artículo.
origin_store_id	Identificador de la sucursal del artículo de origen.
starting_total	Total de existencias iniciales del artículo, al momento de su alta en el almacén.
price_origin	Precio unitario del artículo, al momento de su alta en el almacén.
supplier_id	Identificador del proveedor del artículo.
post_store_id	Identificador de la sucursal destino.
final_total	Total de existencias que se mandaron a la sucursal destino.
price_posting	Precio unitario del artículo, al momento de enviarse a la sucursal destino.
comments	Comentarios acerca del movimiento.
year	Año en que se hizo el movimiento de inventario.
month	Mes en que se hizo el movimiento de inventario.
day	Día en que se hizo el movimiento de inventario.
isSystemRow	Flag que indica si es un registro crítico de la aplicación.
isDeleted	Flag que indica si es un registro eliminado lógicamente.
creationDate	Fecha en que se hizo la operación de ADD del registro.
account_id_creationdate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para el alta de nuevo registro.
updatedate	Fecha en que se hizo la operación de UPDATE del registro (nulo)
account_id_updatedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la actualización.
deletedate	Fecha en que se hizo la operación de DELETE lógico del registro (nulo)
account_id_deletedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la eliminación lógica.

- Catálogo de pedidos (encabezado).

mtOrders (Catálogo de órdenes de compra encabezado).	
Campo:	Descripción:
order_id	Identificador de la compra.
store_id	Identificador de la sucursal donde se realizó la compra.
status_order	Identificador del status de la compra: activa, cancelado o en proceso.
type_order	Tipo de compra: presencial o en línea.
comments	Comentarios acerca de la compra realizada.
issystemrow	Flag que indica si es un registro crítico de la aplicación.
isdeleted	Flag que indica si es un registro eliminado lógicamente.
creationdate	Fecha en que se hizo la operación de ADD del registro.
account_id_creationdate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para el alta de nuevo registro.
updatedate	Fecha en que se hizo la operación de UPDATE del registro (nulo)
account_id_updatedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la actualización.
deletedate	Fecha en que se hizo la operación de DELETE lógico del registro (nulo)
account_id_deletedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la eliminación lógica.

- Catálogo de pedidos (detalle).

mtOrderDetail (Catálogo de órdenes de compra detalle).	
Campo:	Descripción:
row_id	Identificador del registro.
order_id	Identificador de la compra.
sku_id	Identificador del artículo.
quantity	Cantidad de artículos comprados.
price	Precio unitario del artículo.
value_tax	Valor del impuesto aplicado al subtotal del artículo vendido.
total_value	Subtotal del artículo vendido.
issystemrow	Flag que indica si es un registro crítico de la aplicación.
isdeleted	Flag que indica si es un registro eliminado lógicamente.
creationdate	Fecha en que se hizo la operación de ADD del registro.
account_id_creationdate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para el alta de nuevo registro.
updatedate	Fecha en que se hizo la operación de UPDATE del registro (nulo)
account_id_updatedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la actualización.
deletedate	Fecha en que se hizo la operación de DELETE lógico del registro (nulo)
account_id_deletedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la eliminación lógica.

- Catálogo de menú de la aplicación front-end.

vwMenuSystem vwMenuCategories (Lista de opciones de menú).	
Campo:	Descripción:
row_id	Identificador del registro.
parent_id	Identificador del nodo padre.
value_id	Identificador de la opción.
description	Nombre de la opción.
breadcrumb	Ruta de la opción, desde el nodo raíz hasta el nodo hijo.
level	Nivel de nodo, 0 es nodo padre, n-1 es el nodo más interior.
issystemrow	Flag que indica si es un registro crítico de la aplicación.
isdeleted	Flag que indica si es un registro eliminado lógicamente.
creationdate	Fecha en que se hizo la operación de ADD del registro.
account_id_creationdate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para el alta de nuevo registro.
updatedate	Fecha en que se hizo la operación de UPDATE del registro (nulo)
account_id_updatedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la actualización.
deletedate	Fecha en que se hizo la operación de DELETE lógico del registro (nulo)
account_id_deletedate	Identificador de la cuenta de usuario (desde la tabla mtUsers) para la eliminación lógica.

- Catálogo de categorías de artículos (misma estructura que el catálogo anterior).

Obvio: tenemos que crear los scripts de migración fluida a Base de Datos con la aplicación de consola **CA.MigratorDB** y guardar los cambios. La manera de hacerlo, ya está explicado en el capítulo 1 de este documento. Más adelante agregaremos una tabla de clientes, asociar los clientes con sus compras y agregar datos relacionados con el domicilio de las cuentas de usuario, clientes, sucursales y proveedores (estado o entidad federativa, código postal, colonia y municipio).

PASO 2. Separar responsabilidades en los DTO's. Los DTO's que hemos creado, hay que separarlos tambien para cada operación en Base de Datos. Estamos adelantando un patrón de diseño llamado Command And Query Responsibility Segregation (CQRS), cosa que veremos en la siguiente sección. Por ahora, vamos a crear precisamente los DTO's para las operaciones del tipo ADD, UPDATE y DELETE. Los DTO's para la operación READ, siguen sin cambios.

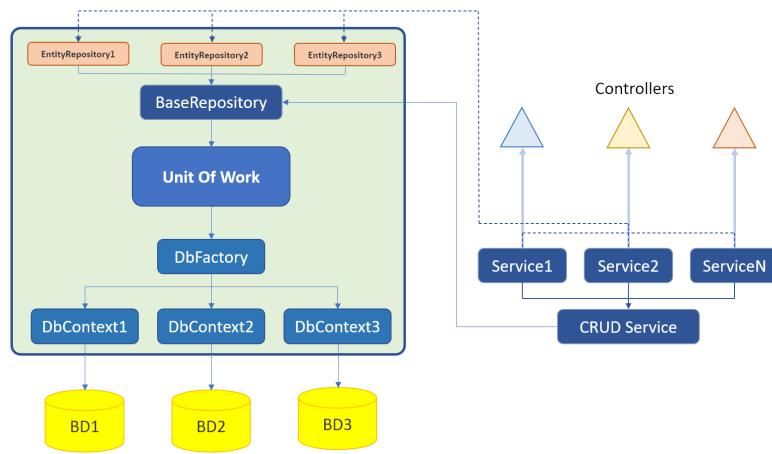
PASO 3. Reforzar la validación fluida. Reforcemos la validación de Fluent Validator, separando los atributos de los DTO's creados para cada una de sus propiedades. Con esto hacemos que nuestro código sea más limpio y facil de mantener.

PASO 4. Aplicar Scaffolding y Fluent API de EntityFramework. Como las tablas se crearon nuevas, hay que realizar los procesos de scaffolding y Fluent API de EntityFramework de nuevo con la nueva estructura de Base de Datos. Esto se explicó ya en las secciones 3 y 4 de este documento.

Teniendo esto, entonces vamos ahora si a proceder a construir nuestra persistencia de datos en base al patrón de diseño Unit Of Work, antes mencionado.

8.11. Diseñando nuestro núcleo de Persistencia de Datos...

Vamos a implementar entonces nuestro modelo de persistencia tomando en cuenta el siguiente diagrama:



Expliquemos el por que usaré este modelo o idea:

- Se crean los contextos que se conectarán directamente a Base de Datos.
- Para cada contexto, se crea una instancia de cada uno de ellos en el objeto del tipo **DbFactory**.
- **UnitOfWork** tomará cada **DbFactory** del contexto de Base de Datos correspondiente.
- Se crea un **BaseRepository** que contendrá la lógica de operaciones sobre la Base de Datos. De este patrón, se crearán los repositorios no genéricos para cada entidad, considerando el contexto de Base de Datos correspondiente.
- Del componente **BaseRepository**, se va a llamar desde un servicio genérico llamado **CRUDService**, donde se realizará toda la lógica del negocio y se hará la transferencia de datos desde un DTO hasta la entidad. Aquí se aplicaría el patrón de diseño Guard Clause.
- De **CRUDService**, a su vez, se apoyarán los servicios, o capa de servicios, apuntando a su respectivo repositorio y su propio contexto de Base de Datos, con el fin de que el flujo de entrada de información apunte respectivamente a su contexto de Base de Datos correspondiente.
- Finalmente, los controllers de la capa mas externa, accederán a los servicios, por medio de la inyección de dependencias.

Con esto tenemos armado el flujo de

8.12. Empezando por la capa de Dominio...

Vamos a implementar entonces nuestra persistencia de datos paso a paso:

1. En la carpeta **CA.Core\DTO**, crear los archivos de clase para los DTO faltantes para las operaciones de agregar, actualizar y eliminar datos, relacionados precisamente para cada objeto entidad de Base de Datos. Por ejemplo, tenemos hasta ahora el DTO llamado **ArticuloDTO**. Tenemos que crear respectivamente los DTO's correspondientes a las demás operaciones de Base de Datos como se muestra a continuación:

CreateArticleDTO.cs:

```
namespace CA.Core.DTO
{
    public class CreateArticleDTO
    {
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public int TotalInVault { get; set; }
        public int DepartmentId { get; set; }
        public int ProductTypeId { get; set; }
        public int SupplierId { get; set; }
        public int BrandId { get; set; }
        public string ImageArticle { get; set; }
        public int AccountIdCreationDate { get; set; }
    }
}
```

UpdateArticleDTO.cs:

```
namespace CA.Core.DTO
{
    public class UpdateArticleDTO
    {
        public int Id { get; set; }
        public double Price { get; set; }
        public int? AccountIdUpdateDate { get; set; }
    }
}
```

DeleteArticleDTO.cs:

```
namespace CA.Core.DTO
{
    public class DeleteArticleDTO
    {
        public int Id { get; set; }
        public bool AutoSave { get; set; }
        public int? AccountIdDeleteDate { get; set; }
    }
}
```

Repetir el mismo procedimiento para las demás entidades de Base de Datos. Nótese que para las operaciones de eliminación, solo se necesitan tres atributos, puesto que haremos eliminación lógica o eliminación física del registro existente en Base de Datos. En el caso del DTO para actualizar, si debemos incluir el identificador del registro y los campos que se quieran actualizar, según sea el caso y la entidad a donde se apunte el DTO. Lo explicaré un poco más adelante.

2. Crear en la carpeta llamada **CA.Core\Interfaces\Management**, los siguientes archivos del tipo interfaz:

IAddEntity.cs

```
using System;
namespace CA.Core.Interfaces.Management
{
    public interface IAddEntity<TKey>
    {
        public TKey Id { get; set; }
        public bool IsSystemRow { get; set; }
        public int AccountIdCreationDate { get; set; }
        public DateTime CreationDate { get; set; }
    }
}
```

IUpdateEntity.cs

```
using System;
namespace CA.Core.Interfaces.Management
{
    public interface IUpdateEntity<TKey> : IAddEntity<TKey>
    {
        public DateTime? UpdateDate { get; set; }
        public int? AccountIdUpdateDate { get; set; }
    }
}
```

IDeleteEntity.cs

```
using System;
namespace CA.Core.Interfaces.Management
{
    public interface IDDeleteEntity<TKey> : IAddEntity<TKey>
    {
        public bool IsDeleted { get; set; }
        public DateTime? DeleteDate { get; set; }
        public int? AccountIdDeleteDate { get; set; }
    }
}
```

IEntityBase.cs

```
using System;
namespace CA.Core.Interfaces.Management
{
    public interface IEntityBase<TKey> : IAddEntity<TKey>, IUpdateEntity<TKey>, IDDeleteEntity<TKey> { }
}
```

Estos archivos de interfaz, manejan las propiedades de agregar, actualizar y eliminar registros. La interfaz base es precisamente **IAddEntity.cs** puesto que ahí se va a definir el identificador del registro y las demás interfaces heredan los atributos desde **IAddEntity.cs**. La interfaz final que se va usar para asignarselo a las entidades de Base de Datos es precisamente **IEntityBase.cs** puesto que reune todo lo que las demás interfaces definen previamente. Aquí estamos aplicando precisamente el DRY principle, mencionado anteriormente.

- Crear una carpeta llamada **CA.Core\Entities\Base** y ahí creamos un archivo de clase llamado **EntityBase.cs**. En este archivo implementará finalmente los atributos de la abstracción **IEntityBase**:

```
using System;
namespace CA.Core.Entities.Base
{
    public abstract class EntityBase<TKey> : IEntityBase<TKey>
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public TKey Id { get; set; }
        public bool IsSyncNow { get; set; }
        public int AccountIdCreationDate { get; set; }
        public DateTime CreationDate { get; set; }
        public DateTime? UpdatedDate { get; set; }
        public int? AccountIdUpdateDate { get; set; }
        public bool IsDeleted { get; set; }
        public DateTime? DeletedDate { get; set; }
        public int? AccountIdDeleteDate { get; set; }
    }
}
```

Aquí, si usaremos **DataAnnotations**, puesto que el atributo Id, lo tenemos que definir como clave principal y como generador de valores de identidad en Base de Datos.

- Guardemos los cambios y ahora, tenemos que las entidades de Base de Datos hereden de EntityBase. Para esto haremos lo siguiente: en la entidad llamada Article.cs, aplicaremos el siguiente cambio y quitemos finalmente los atributos o propiedades que no deben de estar en la entidad en sí, sino solamente las propiedades no comunes de la misma:

```
33 referencias | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 2 cambios
public partial class Article : EntityBase<int>
{
    0 referencias | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
    public Article()
    {
        OrderDetails = new HashSet<OrderDetail>();
        StockInventories = new HashSet<StockInventory>();
    }

    1 referencia | Olimpo Bonilla Ramírez, Hace 74 días | 1 autor, 1 cambio
    public string Name { get; set; }
    1 referencia | Olimpo Bonilla Ramírez, Hace 74 días | 1 autor, 1 cambio
    public string Description { get; set; }
    1 referencia | Olimpo Bonilla Ramírez, Hace 74 días | 1 autor, 1 cambio
    public decimal Price { get; set; }
    1 referencia | Olimpo Bonilla Ramírez, Hace 74 días | 1 autor, 1 cambio
    public int TotalInVault { get; set; }
    1 referencia | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
    public int DepartamentId { get; set; }
    1 referencia | Olimpo Bonilla Ramírez, Hace 74 días | 1 autor, 1 cambio
    public int ProducttypeId { get; set; }
    2 referencias | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
    public int SupplierId { get; set; }
    2 referencias | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
    public int BrandId { get; set; }
    1 referencia | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
    public string ImageArticle { get; set; }

    1 referencia | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
    public virtual User AccountIdCreationDateNavigation { get; set; }
    1 referencia | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
    public virtual Brand Brands { get; set; }
    1 referencia | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
    public virtual Supplier Suppliers { get; set; }
    2 referencias | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
    public virtual ICollection<OrderDetail> OrderDetails { get; set; }
    2 referencias | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
    public virtual ICollection<StockInventory> StockInventories { get; set; }
}
```

Repitamos el mismo procedimiento para las demás entidades de Base de Datos y guardemos los cambios.

- Crear una carpeta llamada **Base** en la carpeta **CA.Core\Interfaces**. Aquí definiremos las abstracciones de los repositorios para generar el repositorio base llamado IBaseRepository:

IReadRepository.cs.

```
namespace CA.Core.Interfaces.Base
{
    3 referencias | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
    public interface IReadRepository<T, TContext>
    {
        where T : class
        where TContext : DbContext, new()

        9 referencias | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
        Task<IEnumerable<T>> AllAsync(CancellationToken cancellationToken = default);
        11 referencias | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
        Task<T> GetByIdAsync(int id, CancellationToken cancellationToken = default);
        9 referencias | Olimpo Bonilla Ramírez, Hace 34 días | 1 autor, 1 cambio
        Task<IEnumerable<T>> FilterAsync(Expression<Func<T, bool>> predicate, CancellationToken cancellationToken = default);
    }
}
```

IBaseRepository.cs.

```

namespace CA.Core.Interfaces.Base
{
    public interface IBaseRepository<T, TContext> : IReadRepository<T, TContext>
        where T : class
        where TContext : DbContext, new()
    {
        Task AddAsync(T Entity, CancellationToken cancellationToken = default);
        void Update(T Entity);
        void Delete(T entity);
    }
}

```

IDbFactory.cs.

```

namespace CA.Core.Interfaces.Base
{
    public interface IDbFactory<TContext> : IDisposable where TContext : DbContext, new()
        where TContext : DbContext, new()
    {
        TContext Init();
    }
}

```

IUnitOfWork.cs.

```

namespace CA.Core.Interfaces.Base
{
    public interface IUnitOfWork<TContext> where TContext : DbContext, new()
    {
        void CreateTransaction();
        Task CreateTransactionAsync();
        Task CreateTransactionAsync(CancellationToken cancellationToken = default);
        void Rollback();
        Task RollbackAsync();
        Task RollbackAsync(CancellationToken cancellationToken = default);
        void Commit();
        Task CommitAsync(CancellationToken cancellationToken = default);
        Task CommitAsync(bool acceptAllChangesOnSuccess, CancellationToken cancellationToken = default);
    }
}

```

Esta es la columna vertebral de Unit Of Work para implementarlo en la capa de Infraestructura. Explicaré más adelante su implementación de cada uno de ellos a detalle. Nótese que se están inyectando en estas interfaces, el tipo genérico llamado `TContext`, el cual, cuando se inyecte, se creará una nueva instancia del contexto de Base de Datos `DbContext` de Entity Framework dentro de su implementación.

- Guardemos los cambios. Ahora crearemos otra carpeta llamada **Repository**. Dentro de esta carpeta, crearemos ahora si los repositorios no genericos para cada entidad. Por ejemplo, para el catálogo de artículos, la abstracción (o interfaz) sería así, en el archivo **IArticleRepository.cs**:

```

namespace CA.Core.Interfaces.Repository
{
    public interface IArticleRepository<TContext> : IBaseRepository<Article, TContext> where TContext : DbContext, new()
    {
        Task<IEnumerable<Article>> GetArticlesAsync(CancellationToken cancellationToken = default);
        Task<Article> GetArticleAsync(int id, CancellationToken cancellationToken = default);
        Task<IQueryable<Article>> FilterArticleAsync(Expression<Func<Article, bool> predicate, CancellationToken cancellationToken = default);
        Task AddArticleSync(Article obj, CancellationToken cancellationToken = default);
        Task<Article> UpdateArticleSync(Article obj, CancellationToken cancellationToken = default);
        void DeleteArticle(Article obj);
    }
}

```

Aquí estamos inyectando el tipo genérico `TContext`, pero heredamos de `IBaseRepository`, los tipos `Article` y `TContext` y obviamente especificamos que cuando se inyecte un `DbContext`, se crea al momento de hacer la implementación. Debemos definir los prototipos de las funciones que esta interfaz implementará en la clase del tipo Repository final, cosa que veremos más adelante. Para las demás entidades de Base de Datos, repitamos este patrón al estilo DRY.

- Guardemos los cambios y finalmente, crearemos otra carpeta llamada **Services**. Dentro de ella, creamos una carpeta llamada **Base** y tendrá dos archivos importantes:

ICRUDService.cs:

```
namespace CA.Core.Interfaces.Services.Base
{
    public interface ICRUDService<TGetDto, TAddDto, TUdpDto, TDelDto, TKey, TEntity, TRepoAll, TContext, TUnitOfWork>
    {
        where TEntity : class, IEntityBase<TKey>;
        where TRepoAll : IRepository<TEntity, TContext>;
        where TContext : DbContext, new()
    }

    5 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
    Task<IEnumerable<TGetDto>> GetAll(CancellationToken cancellationToken = default);
    Task<TGetDto> FindAsync(int id, CancellationToken cancellationToken = default);
    Task<IEnumerable<TGetDto>> FilterAsync(Expression<Func< TEntity, bool> predicate, CancellationToken cancellationToken = default);
    Task<TUdpDto> UpdateAsync(TUdpDto objDTO, CancellationToken cancellationToken = default);
    Task<TAddDto> InsertAsync(TAddDto objDTO, CancellationToken cancellationToken = default);
    Task<TDelDto> DeleteAsync(TDelDto objDTO, bool autoSave = true, CancellationToken cancellationToken = default);
    }
}
```

IRService.cs:

```
namespace CA.Core.Interfaces.Services.Base
{
    public interface IRService<TGetDto, TKey, TEntity, TRepoRead, TContext, TUnitOfWork>
    {
        where TEntity : class, IEntityBase<TKey>;
        where TRepoRead : IReadRepository<TEntity, TContext>;
        where TContext : DbContext, new()
    }

    3 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
    Task<IEnumerable<TGetDto>> GetAll(CancellationToken cancellationToken = default);
    Task<TGetDto> FindAsync(int id, CancellationToken cancellationToken = default);
    Task<IEnumerable<TGetDto>> FilterAsync(Expression<Func< TEntity, bool> predicate, CancellationToken cancellationToken = default);
    1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
    }
}
```

Nótese que en ambas interfaces, tenemos lo siguiente:

- Estamos inyectando los tipos genéricos DTO para todas las operaciones CRUD de **ICRUDService.cs**. Después el tipo genérico de identificador del registro **TKey**, es decir, que acepte un tipo de datos, la entidad de Base de Datos que es **TEntity**, el repositorio que sería en este caso el repositorio base que es **TRepoAll** desde **IBaseRepository**, el contexto de Base de Datos **TContext** y la instancia de UnitOfWork, que viene representada por **TUnitOfWork**.
- Para el caso de la interfaz **IRService**, solo usaremos operaciones de lectura de datos y los objetos genéricos de entrada serían el DTO de lectura de datos, el tipo genérico de identificador del registro **TKey**, la entidad de Base de Datos **TEntity**, el repositorio base de lectura que es **TRepoRead** desde **IReadRepository**, el contexto de Base de Datos **TContext** y la instancia de UnitOfWork, que viene representada por **TUnitOfWork**.

Estas abstracciones de la interfaz, las aplicaremos en la capa de Infraestructura.

8. En la carpeta **CA.Core\Interfaces\Services**, crearemos las abstracciones de los servicios para cada entidad de Base de Datos. Por ejemplo, para la entidad de Base de Datos de artículos, que es **IArticleService.cs**, su definición sería así:

```
namespace CA.Core.Interfaces.Services
{
    4 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
    public interface IArticleService
    {
        2 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        Task<ArticleDTO> FindArticleAsync(int id, CancellationToken cancellationToken = default);
        2 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        Task<IQueryable<ArticleDTO>> GetArticles(CancellationToken cancellationToken = default);
        2 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        Task<CreateArticleDTO> InsertArticleAsync(CreateArticleDTO objDTO, CancellationToken cancellationToken = default);
        2 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        Task<UpdateArticleDTO> UpdateArticleAsync(UpdateArticleDTO objDTO, CancellationToken cancellationToken = default);
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        Task<DeleteArticleDTO> DeleteArticleAsync(DeleteArticleDTO objDTO, bool autoSave = true, CancellationToken cancellationToken = default);
    }
}
```

Repitamos esto mismo para las demás entidades de Base de Datos faltantes. Aquí ya podemos implementar las operaciones CRUD de manera independiente, sin depender del repositorio genérico o no genérico.

9. Finalmente, creamos una carpeta llamada Exceptions y creamos un objeto que herede la clase **Exception** y la llamaremos **EntityNotFoundException.cs**. Esto tiene como finalidad que, cuando se busque un registro en Base de Datos por un identificador de registro, si este no existe... provoque la excepción y se termina el proceso. Su estructura es la siguiente:

```
namespace CA.Core.Exceptions
{
    11 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
    public class EntityNotFoundException : Exception
    {
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public Type EntityType { get; set; }
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public object Id { get; set; }

        0 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public EntityNotFoundException() : base() { }

        0 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public EntityNotFoundException(Type entityType, object id) : this(entityType, null, null) { }

        3 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public EntityNotFoundException(Type entityType, object id, Exception innerException): base(
            id == null ? $"There is no such an entity given given id. Entity type: {entityType.FullName}" :
            $"There is no such an entity. Entity type: {entityType.FullName}, id: {id}", innerException)
        {
            EntityType = entityType; Id = id;
        }

        0 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public EntityNotFoundException(string message) : base(message) { }

        0 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public EntityNotFoundException(string message, Exception innerException) : base(message, innerException) { }
    }
}
```

10. Creamos una nueva carpeta llamada **Wrappers** y en ella vamos a crear una clase llamada **ApiResponse.cs**, cuya estructura es la siguiente:

```
namespace CA.Core.Wrappers
{
    24 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
    public class ApiResponse<T>
    {
        0 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public ApiResponse()
        {
        }

        21 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public ApiResponse(T data, string message = null)
        {
            Succeeded = true;
            Message = message;
            Data = data;
        }

        0 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public ApiResponse(string message)
        {
            Succeeded = false;
            Message = message;
        }

        2 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public bool Succeeded { get; set; }
        2 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public string Message { get; set; }
        0 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public List<string> Errors { get; set; }
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public T Data { get; set; }
    }
}
```

Lo que estamos haciendo aquí es ya implementar un objeto del tipo **Response** que muestre de manera sencilla el resultado final de los request que hagamos en nuestra Web API. Esta clase la vamos a evolucionar más adelante. Mientras, aquí vemos que **ApiResponse** nos proporciona información de como se comporta el **Request** que se envía y la lista de errores, si los hay.

11. Guardemos cambios y no olvidemos incluir las librerías en nuestro proyecto de Domain: [Microsoft.EntityFrameworkCore](#) y [System.ComponentModel.Annotations](#) para compilar este proyecto y evitar errores de compilación.

8.13. Empezando por la capa de Infraestructura.

Teniendo entonces lista la capa interior, que es la de Dominio, empecemos a ajustar y aplicar los cambios a la capa de Infraestructura. Para esto empecemos como sigue:

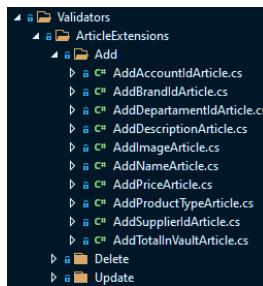
1. En la carpeta de **CA.Infrastructure\Extensions**, creamos una clase estática llamada **RegexExtensions.cs** y su propósito es validar valores bajo patrones de validación del tipo Regex. Esto es importante para reforzar la validación de datos por el componente Fluent Validation para los DTO's, cosa que explicaré más adelante. La estructura es esta:

```
namespace CA.Infrastructure.Extensions.Base
{
    {
        43 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
    }
    public static class RegexExtensions
    {
        {
            43 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
            public static bool VerifyValue(object value, string pattern) => Regex.IsMatch(value.ToString(), pattern);
            0 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
            public static bool VerifyStringIsNullOrEmpty(string value) =>
                (Regex.IsMatch(value, @"^\s*$") | string.IsNullOrEmpty(value) | value.Length == 0 | value == "null" | value == "NULL");
        }
    }
}
```

2. En la carpeta **CA.Infrastructure\Validators**, vamos a extender las validaciones sobre los objetos DTO para reforzar las operaciones CRUD en Base de Datos. Empecemos con los DTO's orientados a los artículos. Para la operación de alta de un nuevo artículo, crearemos una clase llamada CreateArticleValidator en el archivo **CreateArticleValidator.cs**. Su estructura es la siguiente:

```
namespace CA.Infrastructure.Validators
{
    {
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
    }
    public class CreateArticleValidator : AbstractValidator<CreateArticleDTO>
    {
        {
            0 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
            public CreateArticleValidator()
            {
                Include(new AddNameArticle());
                Include(new AddDescriptionArticle());
                Include(new AddPriceArticle());
                Include(new AddTotalInVaultArticle());
                Include(new AddDepartamentIdArticle());
                Include(new AddProductTypeArticle());
                Include(new AddSupplierIdArticle());
                Include(new AddBrandIdArticle());
                Include(new AddImageArticle());
                Include(new AddAccountIdArticle());
            }
        }
    }
}
```

Vamos a separar, o mejor dicho extender las propiedades del DTO **CreateArticleDTO**, creado en **CA.Core\DTO**, en clases separadas. Para esto, creamos una carpeta llamada **ArticleExtensions** y dentro de ella otras carpetas más: **Add, Update y Delete**, respectivamente. En la carpeta **Add**, hay que crear las siguientes clases:



Cada clase, va a tener una validación por Regex, y usaremos entonces la función creada recientemente en **CA.Infrastructure\Extensions**, de la clase estática **RegexExtensions.cs**. Por ejemplo, para validar el campo Description del objeto **CreateArticleDTO**, su validación sería en el archivo **AddDescriptionArticle.cs** y es algo como esto:

```
namespace CA.Infrastructure.Validators
{
    2 referencias | Ólmpo Bonilla Ramrez; Hace 35 días | 1 autor, 1 cambio
    public class AddDescriptionArticle : AbstractValidator<CreateArticleDTO>
    {
        1 referencia | Ólmpo Bonilla Ramrez; Hace 35 días | 1 autor, 1 cambio
        public AddDescriptionArticle()
        {
            RuleFor(u => u.Description).Cascade(CascadeMode.Stop)
                .NotBeNull().WithMessage("La descripción del artículo no puede ser nula.")
                .NotEmpty().WithMessage("La descripción del artículo no puede ser vacío.")
                .Must(u => RegexExtensions.VerifyValue(u, @"^[\w\s]{2,255}$")).WithMessage("Formato de la descripción del artículo incorrecto.");
        }
    }
}
```

Para los demás atributos del objeto **CreateArticleDTO**, aplicar el mismo criterio. Por ejemplo:

```
namespace CA.Infrastructure.Validators
{
    2 referencias | Ólmpo Bonilla Ramrez; Hace 35 días | 1 autor, 1 cambio
    public class AddPriceArticle : AbstractValidator<CreateArticleDTO>
    {
        1 referencia | Ólmpo Bonilla Ramrez; Hace 35 días | 1 autor, 1 cambio
        public AddPriceArticle()
        {
            RuleFor(u => u.Price).Cascade(CascadeMode.Stop)
                .GreaterThanOrEqualTo(0).WithMessage("El precio del artículo no puede ser negativo o cero.")
                .Must(u => RegexExtensions.VerifyValue(u, @"^[\d]{1,13}(\.[\d]{1,2}){0,1}$")).WithMessage("Formato de número decimal incorrecto: 13 dígitos de mantisa y 2 posiciones decimales.");
        }
    }
}
```

Como tenemos un atributo del tipo string que guarda las imágenes en Base64, que es en este caso **ImageArticle**, es necesario tambien realizar un regex para validar que el string en ese formato sea correcto. El patrón de validación de Regex para un archivo de imagen en Base64 sería algo como esto:

```
namespace CA.Infrastructure.Validators
{
    2 referencias | Ólmpo Bonilla Ramrez; Hace 35 días | 1 autor, 1 cambio
    public class AddImageArticle : AbstractValidator<CreateArticleDTO>
    {
        1 referencia | Ólmpo Bonilla Ramrez; Hace 35 días | 1 autor, 1 cambio
        public AddImageArticle()
        {
            RuleFor(u => u.ImageArticle).Cascade(CascadeMode.Stop)
                .NotBeNull().WithMessage("La cadena en Base 64 de la imagen del artículo no puede ser nula.")
                .NotEmpty().WithMessage("La cadena en Base 64 de la imagen del artículo no puede ser vacío.")
                .Must(u => RegexExtensions.VerifyValue(u, @"^data:image/(?:gif|png|jpeg|jpg|bmp|webp|svg+xml)(?:\?;charset=utf-8)?;base64,(?:[A-Za-z0-9]+|[+/-])+(?:,[0-9]{1,3})*$")).WithMessage("Formato de archivo o imagen descriptiva del artículo incorrecto.");
        }
    }
}
```

Repetir el mismo procedimiento para los demás DTO's específicos para las demas entidades de Base de Datos. Notése que tambien es necesario validar que si introducimos un valor booleano, tambien tenemos que checar que en efecto, si sea un booleano. Esto lo aplicamos en la validación de los objetos **UpdateArticleDTO** y **DeleteArticleDTO**, respectivamente y sería algo como esto:

```
namespace CA.Infrastructure.Validators
{
    2 referencias | Ólmpo Bonilla Ramrez; Hace 35 días | 1 autor, 1 cambio
    public class DeleteAutoSaveArticle : AbstractValidator<DeleteArticleDTO>
    {
        1 referencia | Ólmpo Bonilla Ramrez; Hace 35 días | 1 autor, 1 cambio
        public DeleteAutoSaveArticle()
        {
            RuleFor(u => u.AutoScale).Cascade(CascadeMode.Stop)
                .Must(u => RegexExtensions.VerifyValue(u, @"^(true|false)$")).WithMessage("Formato de valor booleano incorrecto: solo 'true' o 'false'.");
        }
    }
}
```

El resto de la implementación de las validaciones por Regex, ya va dependiendo del tipo de dato y la configuración regional correspondientes. Aplicar este mismo criterio para los DTO's que apunten a las operaciones CRUD sobre las entidades de Datos.

- Guardemos cambios y vayamos en la carpeta **CA.Infrastructure\Mappings** y completemos la conversión de los DTO's por el componente de AutoMapper. El archivo donde tenemos que hacer estas adecuaciones es precisamente **AutoMapperProfile.cs**:

```

namespace CA.Infrastructure.Mappings
{
    public class AutoMapperProfile : Profile
    {
        public AutoMapperProfile()
        {
            /* Articulos. */
            CreateMap<Article, ArticleDTO>().ReverseMap();
            CreateMap<CreateArticleDTO, Article>().ReverseMap();
            CreateMap<UpdateArticleDTO, Article>().ReverseMap();
            CreateMap<DeleteArticleDTO, Article>().ReverseMap();

            /* Sucursales. */
            CreateMap<Store, StoreDTO>().ReverseMap();
            CreateMap<CreateStoreDTO, Store>().ReverseMap();
            CreateMap<UpdateStoreDTO, Store>().ReverseMap();
            CreateMap<DeleteStoreDTO, Store>().ReverseMap();

            /* Marcas de articulo. */
            CreateMap<BrandDTO, Brand>().ReverseMap();
            CreateMap<CreateBrandDTO, Brand>().ReverseMap();
            CreateMap<UpdateBrandDTO, Brand>().ReverseMap();
            CreateMap<DeleteBrandDTO, Brand>().ReverseMap();

            /* Proveedores. */
            CreateMap<SupplierDTO, Supplier>().ReverseMap();
            CreateMap<CreateSupplierDTO, Supplier>().ReverseMap();
            CreateMap<UpdateSupplierDTO, Supplier>().ReverseMap();
            CreateMap<DeleteSupplierDTO, Supplier>().ReverseMap();

            /* Categorías. */
            CreateMap<CategoryDTO, MenuCategory>().ReverseMap();
            CreateMap<MenuDTO, MenuSystem>().ReverseMap();

            /* Usuarios (Pendiente...). */
        }
    }
}

```

Con esto, tenemos ahora si reforzado, la conversión de tipos de dato DTO a entidades de Base de Datos. Conforme vaya creciendo la Web API, se van introduciendo más tipos genéricos de DTO y las entidades de Datos correspondientes. Pero vemos que las operaciones CRUD ya están mas separadas y cumplimos el primer principio de SOLID.

4. Guardemos los cambios y crearemos ahora una nueva carpeta llamada Persistence. Dentro de esta carpeta tendremos otras carpetas internas más: **Base**, **Data**, **Repository** y **Services**. Empecemos a crear código fuente en cada una de ellas.
5. En la carpeta **CA.Infrastructure\Persistence\Base**, hay que implementar **UnitOfWork** y **DbFactory** de las interfaces propiamente dichas desde **CA.Core**. La estructura de ambos archivos son los siguientes:

DbFactory.cs

```

namespace CA.Infrastructure.Persistence.Base
{
    public class DbFactory<TContext> : IDisposable, IDbFactory<TContext>
        where TContext : DbContext, new()
    {
        private bool _disposed;
        private TContext _dbContext;
        private Func<TContext> _instanceFunc;

        public DbFactory(Func<TContext> dbContextFactory) => _instanceFunc = dbContextFactory;

        public void Dispose()
        {
            if (!_disposed && _dbContext != null)
            {
                _disposed = true; _dbContext.Dispose(); GC.SuppressFinalize(this);
            }
        }

        public TContext Init() => _dbContext ??= _instanceFunc.Invoke();
    }
}

```

Al inicializarse el objeto del tipo DbFactory (en la función `Init()`), se crea una instancia del contexto de Base de Datos de manera independiente y está lista para ser usada. Cuando ya no se crea necesario usarse, se destruye con `Dispose()`.

UnitOfWork.cs

```
namespace CA.Infrastructure.Persistence.Base
{
    2 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
    public class UnitOfWork<TContext> : IUnitOfWork<TContext>
        where TContext : DbContext, new()
    {
        private TContext _dbContext;
        private IDbContextTransaction _objTran;
        private readonly IDbFactory<TContext> _dbFactory;

        0 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public UnitOfWork(IDbFactory<TContext> dbFactory) => _dbFactory = dbFactory;
        6 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public TContext DbContext
        {
            get { return _dbContext ?? (_dbContext = _dbFactory.Init()); }
        }

        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public void Commit() => DbContext.SaveChanges();
        4 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public async Task CommitAsync(CancellationToken cancellationToken = default) =>
            await DbContext.SaveChangesAsync(cancellationToken);
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public async Task CommitAsync(bool acceptAllChangesOnSuccess, CancellationToken cancellationToken = default) =>
            await DbContext.SaveChangesAsync(acceptAllChangesOnSuccess, cancellationToken);
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public void CreateTransaction() => _objTran = DbContext.Database.BeginTransaction();
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public void Rollback() { _objTran.Rollback(); _objTran.Dispose(); }
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public async Task CreateTransactionAsync() => _objTran = await DbContext.Database.BeginTransactionAsync();
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public async Task RollbackAsync() { await _objTran.RollbackAsync(); await _objTran.DisposeAsync(); }
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public async Task CreateTransactionAsync(CancellationToken cancellationToken = default) =>
            _objTran = await DbContext.Database.BeginTransactionAsync(cancellationToken);
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public async Task RollbackAsync(CancellationToken cancellationToken = default)
        {
            await _objTran.RollbackAsync(cancellationToken); await _objTran.DisposeAsync();
        }
    }
}
```

Nótese lo siguiente: se crean las operaciones principales sobre el objeto del tipo `IDbContextTransaction`, que hace, que todas las operaciones en Base de Datos se hagan en transacciones. Se inyecta el tipo `DbContext`, con lo cual, hace que cualquier contexto de Base de Datos pueda trabajar sus operaciones por cada instancia de `UnitOfWork`.

6. Guardemos los cambios. El contexto de Base de Datos de PatosaComercial y las configuraciones de las entidades de la Base de Datos se mantienen en la carpeta **Data**. Solo hay que hacer los ajustes de las propiedades en base al tipo `EntityBase<T>` para que queden emparejadas. Esto es la parte de Fluent API para EntityFramework.
7. En la carpeta **CA.Infrastructure\Persistence\Repository**, quitemos los archivos de repositorio que estaban originalmente y creamos una carpeta llamada **Base**. Ahí vamos a crear el módulo **BaseRepository.cs** y cuya estructura es la siguiente:

```

namespace CA.Infrastructure.Persistence.Repository.Base
{
    ...
}

```

Este código es una implementación genérica de un repositorio que hereda de `IBaseRepository<T, TContext>`. El constructor recibe un contexto de base de datos (`TContext : DbContext`) y un fabrica de base de datos (`IDbFactory<TContext> dbFactory`). El repositorio tiene métodos para agregar, eliminar y actualizar entidades genericas (`T`).

Vemos que los parámetros que se inyectan son: el tipo genérico `T`, el tipo de dato `TKey` y el contexto de Base de Datos `TContext`, y hereda desde `IBaseRepository<T, TContext>`. Nótese que en el constructor de esta clase se inyecta el objeto `IDbFactory` con cualquier contexto de Base de Datos para activar obviamente el contexto de Base de Datos y el tipo de dato `DbSet<T>` correspondiente a la entidad de Base de Datos. Lo demás ya lo sabemos: las operaciones CRUD de Base de Datos.

- Construida esta implementación del repositorio genérico, construimos sobre él, la implementación de los repositorios no genéricos para cada objeto entidad. Por ejemplo, para el caso de la entidad `Article`, su implementación sería así en el archivo `ArticleRepository.cs`:

```

namespace CA.Infrastructure.Persistence.Repository
{
    ...
}

```

`ArticleRepository` hereda de `BaseRepository` y de `IArticleRepository`. Aquí, ya podemos escribir el contexto de Base de Datos que le corresponde, que es en este caso, `PatosaDbContext`. Esta es la ventaja en la cual, dejamos que el repositorio genérico apunte a cualquier contexto de Base de Datos y no que dependa de uno solo. Lo demás que se inyecta es la entidad `Article` del contexto de Base de Datos y el tipo de dato para el identificador del registro, que es el tipo `int`. Repitamos esto para las demás entidades de Base de Datos, aplicando esta implementación para cada uno de los repositorios no genéricos y definiendo correctamente su entidad de Base de Datos correspondiente.

- Guardemos los cambios y ahora pasemos a generar el servicio general llamado `CRUDService.cs` y lo vamos a generar en la carpeta nueva llamada `Base` de la carpeta `CA.Infrastructure\Persistence\Services`. Su estructura es la siguiente:

```


namespace CD.Infrastructure.Persistence.Services.Base
{
    [ExcludeFromCodeCoverage]
    public abstract class CRUDService<TGetDto, TAddDto, TUpdDto, TKey, TEntity, TRepAll, TContext, TUnitOfWork> : ICUDService<TGetDto, TAddDto, TUpdDto, TKey, TEntity, TRepAll, TContext, TUnitOfWork>
    where TEntity : class, IEntity<TKey>
    where TGetDto : class, IEntity<TKey>, new()
    where TRepAll : IRepAll<TEntity, TContext>
    where TContext : IContext<TEntity>
    where TUnitOfWork : IUnitOfWork<TContext>

    {
        internal readonly IMapper _mapper;
        internal readonly IRepAll<TRepAll> _repository;
        internal readonly IUnitOfWork<TUnitOfWork> _unitOfWork;
        protected readonly GuardedMapper _guardedMapper;
        protected readonly GuardedRepository<TRepAll> _repositoryGuarded;
        protected readonly GuardedMapper _unitOfWorkGuarded;
        protected readonly IUnitWorkContexts<TUnitOfWork> _unitOfWork;

        public CRUDService(IRepAll<TRepAll> repository, IUnitWorkContexts<TUnitOfWork> unitOfWork, IMapper mapper)
        {
            _repository = Guard.Against.Null(repository, nameof(repository));
            _mapper = Guard.Against.Null(mapper, nameof(mapper));
            _unitOfWork = Guard.Against.Null(unitOfWork, nameof(unitOfWork));
        }

        [ExcludeFromCodeCoverage]
        public async Task<TDelDto> DeleteSync(TDelDto objJTO, bool autoSave = true, CancellationToken cancellationToken = default);
        [ExcludeFromCodeCoverage]
        public async Task<TDelDto> FindSync<T>(int id, CancellationToken cancellationToken = default);
        [ExcludeFromCodeCoverage]
        public async Task<TDelDto> GetSync<T>(T objJTO, CancellationToken cancellationToken = default);
        [ExcludeFromCodeCoverage]
        public async Task<TDelDto> InsertSync<T>(T objJTO, CancellationToken cancellationToken = default);
        [ExcludeFromCodeCoverage]
        public async Task<TDelDto> UpdateSync<T>(T objJTO, CancellationToken cancellationToken = default);
        [ExcludeFromCodeCoverage]
        public async Task<TDelDto> UpdateSync<T>(T objJTO, FilterAsync<ExpressionFunc<TEntity, bool>> predicate, CancellationToken cancellationToken = default);
    }
}


```

Notemos que se inyectan, desde la interfaz **ICRUDService**:

- **TGetDto**: Un DTO para el despliegado de datos, o sea, para la operación de lectura de datos.
- **TAddDto**: Un DTO para la operación de agregar un nuevo registro.
- **TUpdDto**: Un DTO para la operación de actualizar un registro existente.
- **TDelDto**: Un DTO para la operación de eliminar un registro existente.
- **TKey**: El tipo de dato para el identificador del registro.
- **TEntity**: El objeto de entidad contenido en el contexto de Base de Datos.
- **TRepoAll**: El repositorio base para todas las operaciones CRUD.
- **TContext**: El contexto de Base de Datos.
- **TUnitOfWork**: El objeto UnitOfWork general, para el contexto de Base de Datos que se está usando.

El constructor de esta clase, se aplica precisamente la inyección de dependencias pero con cláusulas de protección para los objetos del tipo para evitar que estos objetos entren como objetos nulos. Aquí se utiliza el componente Ardalis.GuardClauses para .NET Core y sería esto:

```


public CRUDService(IRepoAll repository, IUnitOfWork<TContext> unitOfWork, IMapper mapper)
{
    _repository = Guard.Against.Null(repository, nameof(repository));
    _mapper = Guard.Against.Null(mapper, nameof(mapper));
    _unitOfWork = Guard.Against.Null(unitOfWork, nameof(unitOfWork));
}


```

La idea de esta clase es concentrar en un solo lugar, el poder realizar los mapeos de los objetos del tipo DTO a las entidades de Base de Datos con AutoMapper y no estar repitiendo esto en cualquier parte del código fuente. Si en cada una de las operaciones CRUD sucede que falle el mapeo, se lanzará una excepción en tiempo de ejecución y se detendrá la operación CRUD que esté presentando la falla en ese instante.

Las operaciones CRUD, ya las conocemos. Queda a criterio del desarrollador, el crear la lógica de operación en Base de Datos.

10. Guardemos los cambios y ahora vamos a crear, otro archivo llamado **RService.cs**. Es similar a **CRUDService**, solo que se enfoca a las operaciones de solo lectura de Base de Datos. Su implementación sería como esto:

```

namespace CA.Infrastructure.Persistence.Services.Base
{
    5 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
    public abstract class RService<IGetDto, TKey, TEntity, TRepoRead, TContext, TUnitOfWork> : IService<IGetDto, TKey, TEntity, TRepoRead, TContext, TUnitOfWork>
    where TEntity : class, IEntityBase< TKey > where TRepoRead : IReadRepository where TContext : DbContext, new()
    {
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        internal readonly IMapper _mapper;
        internal readonly TRepoRead _repository;
        internal readonly IUnitOfWork<TContext> _unitOfWork;

        3 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        protected IMapper Mapper => _mapper;
        3 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        protected TRepoRead Repository => _repository;
        2 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public RService(IRepoRead repository, IUnitOfWork<TContext> unitOfWork, IMapper mapper)
        {
            _unitOfWork = Guard.Against.Null(unitOfWork, nameof(unitOfWork));
            _repository = Guard.Against.Null(repository, nameof(repository));
            _mapper = Guard.Against.Null(mapper, nameof(mapper));
        }

        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public Task<IEnumerable<IGetDto>> FilterAsync(Expression

```

De igualmanera, tambien se aplica la técnica de Guard Clause en el constructor de este objeto. Aquí solamente se inyecta **IGetDto** para desplegar datos. Lo demás es similar.

11. Guardemos esto y ahora si, vamos a crear un módulo llamado **ArticleService.cs** y se construirá sobre **CRUDService.cs**. Su estructura es la siguiente:

```

namespace CA.Infrastructure.Persistence.Services
{
    2 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
    public class ArticleService : CRUDService<ArticleDTO, CreateArticleDTO, UpdateArticleDTO, DeleteArticleDTO, int,
        Article, ArticleRepository<PatosaDbContext>, PatosaDbContext, IUnitOfWork>, IArticleService
    {
        0 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public ArticleService(IMapper mapper, IUnitOfWork<PatosaDbContext> unitOfWork, IArticleRepository<PatosaDbContext> articleRepository) : base(articleRepository, unitOfWork, mapper) { }

        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public async Task<ArticleDTO> FindArticleByIdSync(int id, CancellationToken cancellationToken = default) =>
            await FindAsync(id, cancellationToken);
        2 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public async Task<IEnumerable<ArticleDTO>> GetArticles(CancellationToken cancellationToken = default) =>
            await GetAll(cancellationToken);
        2 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public async Task<ArticleDTO> CreateArticleSync(CreateArticleDTO objDTO, CancellationToken cancellationToken = default) =>
            await InsertAsync(objDTO, cancellationToken);
        2 referencias | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public async Task<UpdateArticleDTO> UpdateArticleSync(UpdateArticleDTO objDTO, CancellationToken cancellationToken = default) =>
            await UpdateAsync(objDTO, cancellationToken);
        1 referencia | Olimpo Bonilla Ramírez, Hace 35 días | 1 autor, 1 cambio
        public async Task<DeleteArticleDTO> DeleteArticleSync(DeleteArticleDTO objDTO, bool autoSave = true, CancellationToken cancellationToken = default) =>
            await DeleteAsync(objDTO, autoSave, cancellationToken);
    }
}

```

Aquí ya podemos inyectar todos los objetos DTO asociados a las operaciones CRUD de la tabla de artículos, es decir, **Article** y estos son: **ArticleDTO**, **CreateArticleDTO**, **UpdateArticleDTO** y **DeleteArticleDTO**, creados pasos antes. Se inyecta el tipo de dato **int** para el identificador del registro, así como tambien la entidad de Base de Datos **Article**, la interfaz del repositorio **IArticleRepository<PatosaDbContext>**, el contexto de Base de Datos y **IUnitOfWork** apuntado al contexto de Base de Datos. No olvidemos que **ArticleService** hereda de **IArticleService** para implementar sus funciones correspondientes.

El constructor lo debemos definirlo así:

```

public ArticleService(IMapper mapper, IUnitOfWork<PatosaDbContext> unitOfWork, IArticleRepository<PatosaDbContext> articleRepository) :
    base(articleRepository, unitOfWork, mapper) {}

```

donde se inyecta:

- Un objeto **IMapper**.
- Un objeto **IUnitOfWork** con el contexto de Base de Datos donde va dirigido.
- Un objeto **IArticleRepository** con el contexto de Base de Datos donde va dirigido.

Repitamos el mismo proceso para los demas servicios para cada entidad de Base de Datos, aplicando el estandar antes explicado y guardemos los cambios.

12. En la carpeta **CA.Infrastructure\ServiceCollection**, vamos a crear un módulo más para que realice la inversión de control para el ciclo de vida del objeto **DbFactory** que hemos creado en la capa de Infraestructura. Este archivo se llama **DbCtxFactory.cs** y su estructura es la siguiente:

```

namespace CA.Infrastructure.Extensions.ServiceCollection
{
    public static class DbCtxFactory
    {
        public static IServiceCollection AddDbFactory(this IServiceCollection services)
        {
            services.AddScoped<Func<PatosaDbContext>>((provider) => provider.GetService<PatosaDbContext>());
            /* Agregar aquí las implementaciones de Factory Pattern, asociadas a cada contexto de Base de Datos... */
            return services;
        }
    }
}

```

Lo que estamos haciendo aquí es crear el ciclo de vida para los objetos del tipo **DbFactory** que contengan los contextos de Base de Datos y que por separado, puedan crear sus instancias de manera independiente. Por eso su ciclo de vida es del tipo **Scoped**.

13. Tenemos que registrar ahora si los servicios y repositorios que hemos creado, así como los objetos del tipo **UnitOfWork** y **DbFactory**. Esto lo haremos en el archivo **IoC.cs** de la carpeta **CA.Infrastructure\Extensions\ServiceCollection**:

```

namespace CA.Infrastructure.Extensions.ServiceCollection
{
    public static class IoC
    {
        public static IServiceCollection AddDependency(this IServiceCollection services)
        {
            /* Factory y Unit of Work. */
            services.AddScoped<IDbFactory<PatosaDbContext>, DbFactory<PatosaDbContext>>();
            services.AddScoped<IUnitOfWork<PatosaDbContext>, UnitOfWork<PatosaDbContext>>();

            /* Repositorios. */
            services.AddTransient<IArticleRepository<PatosaDbContext>, ArticleRepository>();
            services.AddTransient<IStoreRepository<PatosaDbContext>, StoreRepository>();
            services.AddTransient<IBrandRepository<PatosaDbContext>, BrandRepository>();
            services.AddTransient<ISupplierRepository<PatosaDbContext>, SupplierRepository>();
            services.AddTransient<ICategoryRepository<PatosaDbContext>, CategoryRepository>();
            services.AddTransient<IMenuRepository<PatosaDbContext>, MenuRepository>();

            /* Servicios. */
            services.AddTransient<IArticleService, ArticleService>();
            services.AddTransient<IStoreService, StoreService>();
            services.AddTransient<IBrandService, BrandService>();
            services.AddTransient<ISupplierService, SupplierService>();
            services.AddTransient<ICategoryService, CategoryService>();
            services.AddTransient<IMenuService, MenuService>();

            return services;
        }
    }
}

```

Notése que el contexto de Base de Datos, para cada repositorio, debe declararse.

14. En la carpeta
15. Con esto ya tenemos concluida la capa de Infraestructura con la persistencia de datos establecida. Nos falta la capa de presentación para que ya tome los cambios aplicados.

8.14. Terminando la capa de Presentación.

Finalmente ya podemos terminar la parte mas externa de nuestro proyecto y procedemos así:

- En la carpeta **CA.Api\Controllers**, hay que refactorizar cada controller para los catálogos de Base de Datos. Por ejemplo, para **ArticleController.cs**, el cambio sería así:

```

namespace CA.Api.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ArticleController : ControllerBase
    {
        private readonly IArticleService _articleService;

        public ArticleController(IArticleService articleService)
        {
            _articleService = articleService;
        }

        [HttpGet]
        public async Task<IActionResult> GetArticles()
        {
            return Ok(await _articleService.GetArticles());
        }

        [HttpGet("{id}")]
        public async Task<IActionResult> GetArticles(int id)
        {
            var article = await _articleService.GetArticle(id);
            if (article == null)
                return NotFound();
            return Ok(article);
        }

        [HttpPost]
        public async Task<IActionResult> Post(CreateArticleDTO obj)
        {
            obj = await _articleService.InsertArticleAsync(obj);
            var response = new ApiResponse<CreateArticleDTO>(obj);
            return Ok(response);
        }

        [HttpPut]
        public async Task<IActionResult> Put(UpdateArticleDTO obj)
        {
            obj = await _articleService.UpdateArticleAsync(obj);
            var response = new ApiResponse<UpdateArticleDTO>(obj);
            return Ok(response);
        }
    }
}

```

Vemos que ya podemos realizar las demás operaciones CRUD pendientes que nos faltaban: la actualización por el verbo **PUT** y la eliminación por el verbo **DELETE**. Hay que aplicarlo para los demás controllers de cada uno de los catálogos de la aplicación Web API.

2. En el archivo **StartUp.cs**, como hicimos ajustes en la inversión de control, tenemos que incluir las siguientes líneas de código en la función **ConfigureServices**:

```

// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    /* Añadimos "AutoMapper" antes de configurar los controllers y se cargan todas la configuraciones de
     * AutoMapper en las referencias de este ensamblado. */
    /* services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies()); */
    /* Otra forma */
    services.AddAutoMapper(typeof(Startup).Assembly, typeof(AutoMapperProfile).Assembly);

    /* Añadir controllers y configurando JSON para evitar referencias circulares, ignorando atributos nulos y
     * formateando los atributos en notación "CamelCase", así como ajustando la zona horaria a meridiano local. */
    CtrlCfg.AddControllersExtend(services);

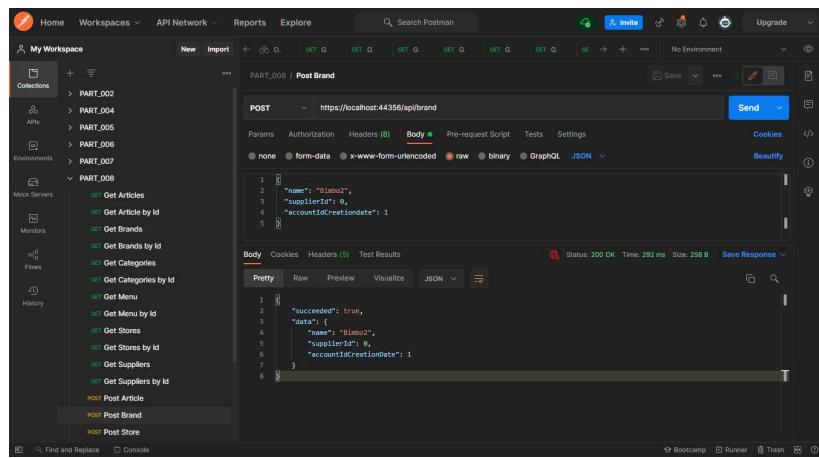
    /* Cadena de conexión al contexto de Base de Datos. */
    DbCtx.AddDbContexts(services, Configuration);

    /* Implementaciones del tipo Factory. */
    DbCtxFactory.AddDbFactory(services);

    /* Contenedor de inversión de control (IoC) => Middleware. */
    IoC.AddDependency(services);
}

```

3. Guardemos los cambios y compilemos. Si todo ha salido bien, nuestra Web API estará funcionando de manera correcta.



- Notemos que ya podemos realizar las operaciones CRUD con la persistencia de datos ya configurada y revisar los resultados en la Base de Datos y probemos su funcionamiento correcto.

Nos faltaría muchas cosas que mejorar para que la persistencia de datos quede completada. Esto sería:

- Control de errores y despliegado de los mismos.
- Separación de las operaciones CRUD por responsabilidad (CSQR).
- Auditoria de movimientos en Base de Datos (Audit Trailing).
- Completar las operaciones CRUD para las compras y agregar un nuevo catálogo de clientes para relacionarlo con sus compras mismas.

Pero con esta implementación, ya tenemos cubierto el pendiente de la construcción de la Persistencia de Datos y la mejora de la validación fluida de los objetos DTO para validar correctamente los datos de entrada.

8.15. Resumen.

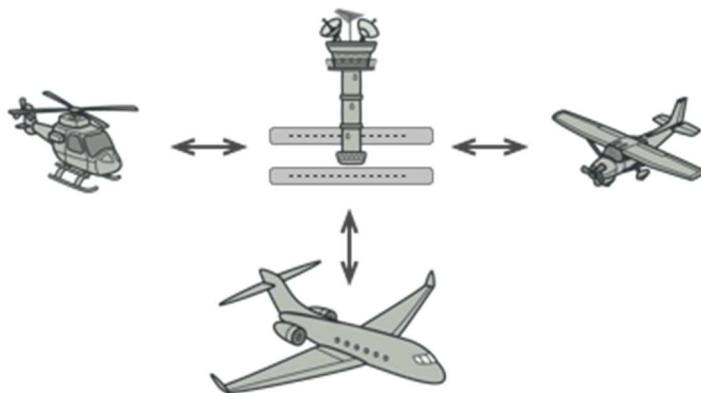
Implementar **UnitOfWork** y la persistencia de Datos es fundamental para todo desarrollo backend con el fin de eficientar las operaciones de Base de Datos de manera optima, segura y eficaz, aplicando correctamente los patrones de diseño adecuados que nos permitan dar mayor mantenimiento al código fuente.

9. Persistencia de Datos (II): CQRS, Mediator y Control de Excepciones

En esta sección, vamos a ver ahora, uno de los patrones de diseño más importantes que han retomado importancia en estos últimos años en el desarrollo de aplicaciones Backend de cualquier lenguaje de programación y que nos permite separar las operaciones de consulta y comando en capas lógicas distintas, a la hora de hacer operaciones CRUD en Web API. Me refiero a Command Query Responsibility Segregation (Segregación de Responsabilidad de Consulta y Comando) y que es de vital importancia entenderlo para desarrollar backend de manera eficiente y responsable.

9.1. Mediator Pattern.

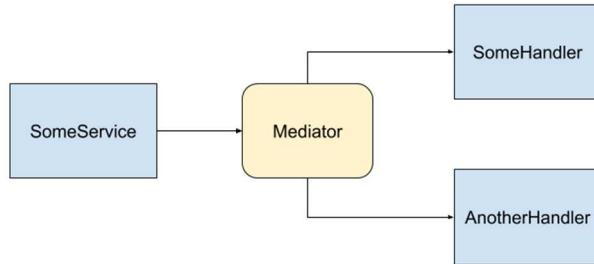
Mediator (mediador) es un patrón de diseño de comportamiento que reduce el acoplamiento entre los componentes de un programa haciendo que se comuniquen indirectamente a través de un objeto mediador especial. El patrón Mediator facilita la modificación, extensión y reutilización de componentes individuales porque ya no son dependientes de todas las demás clases.



*Los pilotos de aviones no hablan directamente entre sí para decidir quién es el siguiente en aterrizar su avión.
Todas las comunicaciones pasan por la torre de control.*

Analogía en el mundo real. Los pilotos de los aviones que llegan o salen del área de control del aeropuerto no se comunican directamente entre sí. En lugar de eso, hablan con un controlador de tráfico aéreo, que está sentado en una torre alta cerca de la pista de aterrizaje. Sin el controlador de tráfico aéreo, los pilotos tendrían que ser conscientes de todos los aviones en las proximidades del aeropuerto y discutir las prioridades de aterrizaje con un comité de decenas de otros pilotos. Probablemente, esto provocaría que las estadísticas de accidentes aéreos se dispararan. La torre no necesita controlar el vuelo completo. Sólo existe para imponer límites en el área de la terminal porque el número de actores implicados puede resultar difícil de gestionar para un piloto.

Otro caso. Esto es a grosso modo, el patrón **Mediator**, el cual, se usa en algunos casos. Pongamos otro caso:



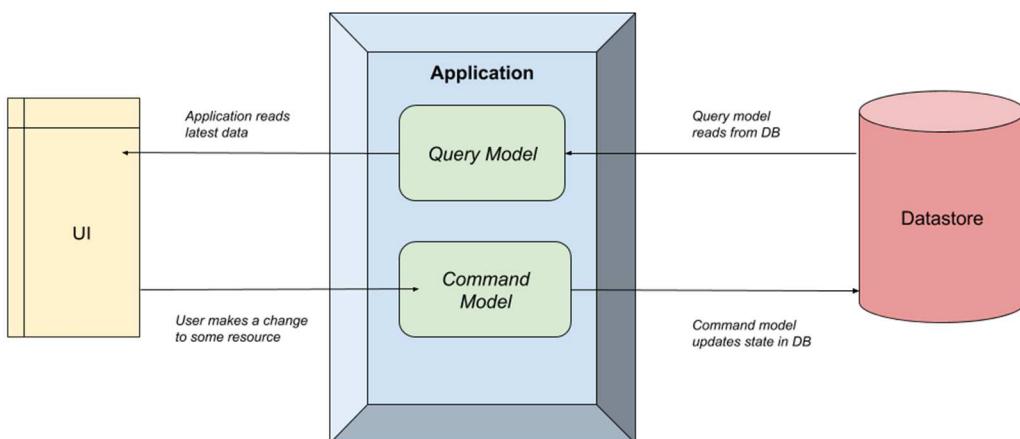
Podemos ver en la imagen de arriba que **SomeService** envía un mensaje a **Mediator**, y éste luego invoca múltiples servicios para manejar el mensaje. No hay dependencia directa entre ninguno de los componentes azules. La razón por la que el patrón Mediator es útil es la misma razón por la que los patrones como Inversion of Control son útiles. Permite el "acoplamiento flexible", ya que el gráfico de dependencia se minimiza y, por lo tanto, el código es más simple y más fácil de probar. En otras palabras, cuantas menos consideraciones tenga un componente, más fácil será su desarrollo y evolución.

Vimos en la imagen anterior como los servicios no tienen dependencia directa, y el productor de los mensajes no sabe quién o cuántas cosas lo van a manejar. Esto es muy similar a cómo funciona un intermediario de mensajes en el patrón de diseño **Publish/Subscribe**. Si quisieramos agregar otro controlador, podríamos, y el productor no tendría que modificarse.

Ahora que hemos repasado algo de teoría, hablemos de cómo MediatR hace que todas estas cosas sean posibles.

9.2. CQRS Pattern.

CQRS significa **Segregación de Responsabilidad de Consultas de Comando (Command Query Responsibility Segregate)**. Como sugiere el acrónimo propuesto por **Greg Young** en 2008, se trata de dividir la responsabilidad de los comandos (guardar) y las consultas (leer) en diferentes modelos. Si pensamos en el patrón CRUD de uso común (Create, Read, Update y Delete), generalmente tenemos la interfaz de usuario interactuando con un almacén de datos responsable de las cuatro operaciones. En cambio, CQRS nos haría dividir estas operaciones en dos modelos, uno para las consultas (también conocido como "Read") y otro para los comandos (también conocido como "Create", "Update" y "Delete"). La siguiente imagen ilustra cómo funciona esto:



Como podemos ver, la aplicación simplemente separa los modelos de consulta y comando. El patrón **CQRS** no establece requisitos formales sobre cómo se produce esta separación. Podría ser tan simple como una clase

separada en la misma aplicación (como veremos más adelante), hasta aplicaciones físicas separadas en diferentes servidores. Esta decisión se basaría en factores como los requisitos de escala y la infraestructura, por lo que no tomaremos ese camino de decisión hoy. El punto clave es que para crear un sistema CQRS, solo necesitamos dividir las lecturas de las escrituras.

¿Qué problema está tratando de resolver esto? Bueno, una razón común es que cuando diseñamos un sistema, comenzamos con el almacenamiento de datos. Realizamos la normalización de la base de datos, agregamos claves primarias y externas para hacer cumplir la integridad referencial, agregamos índices y, en general, nos aseguramos de que el "sistema de escritura" esté optimizado. Esta es una configuración común para una base de datos relacional como SQL Server o MySQL. Otras veces, primero pensamos en los casos de uso de lectura, luego intentamos agregarlos a una base de datos, preocupándonos menos por la duplicación u otras preocupaciones de bases de datos relacionales (a menudo se usan "bases de datos de documentos" para estos patrones). Ningún enfoque es incorrecto. Pero el problema es que es un acto de equilibrio constante entre lecturas y escrituras, y eventualmente un lado "ganará". Todo desarrollo posterior significa que ambos lados deben ser analizados y, a menudo, uno se ve comprometido.

CQRS nos permite "liberarnos" de estas consideraciones y dar a cada sistema el mismo diseño y consideración que merece, sin preocuparnos por el impacto del otro sistema. Esto tiene enormes beneficios tanto en el rendimiento como en la agilidad, especialmente si equipos separados están trabajando en estos sistemas.

Ventajas y desventajas de CQRS. ¿Cuáles son los beneficios de CQRS y por qué deberíamos considerar usarlo en nuestra aplicación?

Los beneficios de CQRS son:

- **Responsabilidad única.** Los comandos y las consultas tienen un solo trabajo. Es para cambiar el estado de la aplicación o recuperarla. Por lo tanto, son muy fáciles de razonar y comprender.
- **Desacoplamiento.** El comando o consulta está completamente desacoplado de su controlador, lo que le brinda mucha flexibilidad en el lado del controlador para implementarlo de la mejor manera que le parezca.
- **Escalabilidad.** El patrón CQRS es muy flexible en términos de cómo puede organizar su almacenamiento de datos, lo que le brinda opciones para una gran escalabilidad. Puede utilizar una base de datos tanto para Comandos como para Consultas. Puede usar bases de datos de lectura/escritura separadas, para mejorar el rendimiento, con mensajería o replicación entre las bases de datos para la sincronización.
- **Capacidad de prueba.** Es muy fácil probar los controladores de comandos o consultas, ya que serán muy simples por diseño y realizarán un solo trabajo.

Por supuesto, no todo puede ser bueno. Estas son algunas de las desventajas de CQRS:

- **Complejidad.** CQRS es un patrón de diseño avanzado y le llevará tiempo comprenderlo por completo. Introduce mucha complejidad que creará fricción y problemas potenciales en su proyecto. Asegúrese de considerar todo antes de decidir usarlo en su proyecto.
- **Curva de aprendizaje.** Aunque parece un patrón de diseño sencillo, todavía hay una curva de aprendizaje con CQRS. La mayoría de los desarrolladores están acostumbrados al estilo procedimental (imperativo) de escribir código, y CQRS se aleja mucho de eso.
- **Difícil de depurar.** Dado que los comandos y las consultas están desacoplados de su controlador, no existe un flujo imperativo natural de la aplicación. Esto hace que sea más difícil de depurar que las aplicaciones tradicionales.

9.3. MediatR: CQRS y Mediator en un solo lugar.

Existen algunos componentes que manejan los patrones de diseño Mediator y CQRS en .NET Core. En este caso usaremos **MediatR** para .NET Core. MediatR fue creado por Jimmy Bogard en 2012 y tiene como

propósito el desvincular el envío de mensajes en proceso del manejo de mensajes. La manera de conseguirlo desde Nuget es la siguiente:

```
$ dotnet add package MediatR
$ dotnet add package MediatR.Extensions.Microsoft.DependencyInjection
```

Para configurarlo, **MediatR no tiene dependencias**. Deberá configurar un único *Factory Delegate*, que se utiliza para crear instancias de todos los controladores, comportamientos de canalización y preprocesadores y posprocesadores. Se deberá configurar dos dependencias: primero, **el propio mediador**. La otra dependencia es el delegado de fábrica, ServiceFactory. La clase Mediator se define como:

```
public class Mediator : IMediator
{
    public Mediator(ServiceFactory serviceFactory)
```

Los *Factory Delegate* se denominan delegados en torno a un par de métodos de fábrica genéricos.

```
public delegate object ServiceFactory(Type serviceType);
```

Declare el tipo de controlador que necesite: sincronizado, asíncrono o asíncrono cancelable. Desde el lado de **IMediator**, la interfaz es solo asíncrona, diseñada para hosts modernos. Finalmente, deberá registrar sus manipuladores en el contenedor de su elección. En este caso, para nuestro propósito, usaremos la primera opción.

Configuración de MediatR en .NET Core. Para configurarlo en .NET Core, puede omitir la configuración anterior y usar el paquete [MediatR.Extensions.Microsoft.DependencyInjection](#) de MediatR, que incluye un método de extensión `IServiceCollection.AddMediatR(Assembly)`, lo que le permite registrarse todos los controladores y pre/postprocesadores en un ensamblaje dado. Esta es la manera mas sencilla de hacerlo:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddMediatR(typeof(Startup));
}
```

MediatR es muy flexible a la hora de configurarse de diversos modos como Autofac, Castle Windsor, Ninject, etc.

9.4. Uso básico de MediatR.

MediatR tiene dos tipos de mensajes que envía:

- **Mensajes de solicitud/respuesta**, enviados a un solo controlador.
- **Mensajes de notificación**, enviados a múltiples controladores.

Request/Response. La interfaz de Request/Response maneja escenarios de comando y consulta. Primero, crea un mensaje:

```
public class Ping : IRequest<string> { }
```

Después, creamos un objeto del tipo Handler:

```

public class PingHandler : IRequestHandler<Ping, string>
{
    public Task<string> Handle(Ping request, CancellationToken cancellationToken)
    {
        return Task.FromResult("Pong");
    }
}

```

Finalmente, enviamos un mensaje por medio del objeto **IMediator**:

```

public class MyClass
{
    private readonly IMediator mediator;

    public MyClass() { }

    public void SendSample()
    {
        var response = await mediator.Send(new Ping());
        Debug.WriteLine(response); // "Pong"
    }
}

```

Con esto, de manera sencilla, usaremos MediatR. En caso de que su mensaje no requiera una respuesta, use la clase base **AsyncRequestHandler<TRequest>**:

```

public class OneWay : IRequest { }
public class OneWayHandlerWithBaseClass : AsyncRequestHandler<OneWay>
{
    protected override Task Handle(OneWay request, CancellationToken cancellationToken)
    {
        // Twiddle thumbs
    }
}

```

Si es completamente síncrona, hereda de la clase base **RequestHandler**:

```

public class SyncHandler : RequestHandler<Ping, string>
{
    protected override string Handle(Ping request)
    {
        return "Pong";
    }
}

```

Tipos de Peticiones. Hay dos tipos de solicitudes en MediatR: unas que devuelven un valor y otras que no:

- **IRequest<T>**. La solicitud devuelve un valor.
- **IRequest**. La solicitud no devuelve un valor.

Para simplificar la canalización de ejecución, **IRequest** hereda **IRequest<Unit>** donde **Unit** representa un tipo de devolución terminal o ignorado. Cada tipo de solicitud tiene su propia interfaz de controlador, así como algunas clases base auxiliares:

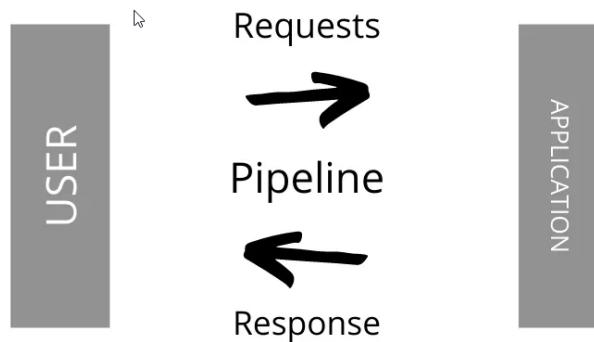
- **IRequestHandler<T, U>**. Implementando esto, devolverá un **Task<U>**.
- **RequestHandler<T, U>**. Heredando esto, devuelve un tipo **U**.

Luego, para solicitudes sin valores de retorno:

- **IRequestHandler<T>**. Implementando esto, devolverá un **Task<Unit>**.
- **AsyncRequestHandler<T>**. Heredando esto, devolverá un **Task**.
- **RequestHandler<T>**. Heredando esto, no devolverá nada (vacío).

9.5. Pipeline en ASP.NET Core.

Vimos en el tema de AutoFilters, a groso modo, el modelo de Pipeline de ASP.NET Core. Volvemos a retomarlo: ¿qué pasa internamente cuando enviamos una solicitud a cualquier aplicación? Lo ideal sería que devuelva la respuesta. Pero hay una cosa de la que quizás ya estemos consciente: **Pipelines**. Ahora, estas solicitudes y respuestas van y vienen a través de Pipelines en ASP.NET Core. Por lo tanto, cuando envía una solicitud, el mensaje de solicitud pasa del usuario a través de una tubería hacia la aplicación, donde realiza la operación solicitada con el mensaje de solicitud. Una vez hecho esto, la aplicación envía el mensaje como respuesta a través de la tubería hacia el usuario final. ¿Lo entiendes? Por lo tanto, estas canalizaciones conocen completamente cuál es la solicitud o la respuesta. Este también es un concepto muy importante mientras se aprende sobre Middlewares en ASP.NET Core. Aquí hay una imagen que representa el concepto antes mencionado.



Digamos que quiero validar el objeto de solicitud. ¿Como lo haremos? Básicamente, escribiríamos las lógicas de validación que se ejecutan después de que la solicitud haya llegado al final de la canalización hacia la aplicación. Eso significa que está validando la solicitud solo después de que haya llegado al interior de la aplicación. Aunque este es un buen enfoque, pensemos al respecto. ¿Por qué necesita adjuntar las lógicas de validación a la aplicación, cuando ya puede validar las solicitudes entrantes incluso antes de que llegue a cualquiera de las lógicas de la aplicación? Un mejor enfoque sería conectar de alguna manera sus lógicas de validación dentro de la canalización, de modo que el flujo se convierta en como si el usuario enviara una solicitud a través de la canalización (aquí las lógicas de validación), si la solicitud es válida, se aplican la lógica de la aplicación, de lo contrario, arroja una excepción de validación. Esto tiene mucho sentido en términos de eficiencia, ¿verdad? ¿Por qué ingresar a la aplicación con datos no válidos, cuando podría filtrarlos mucho antes? Esto no solo se aplica a las validaciones, sino también a otras operaciones, como registro, seguimiento del rendimiento y mucho más.

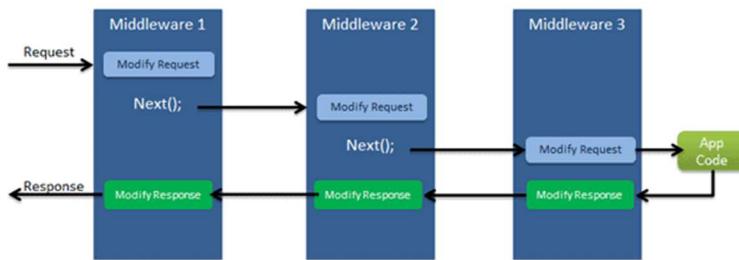
En resumen, la **canalización (Pipelining)** es el proceso de acumulación de instrucciones desde el procesador a través de una canalización. Permite el almacenamiento y ejecución de instrucciones en un proceso ordenado. También se conoce como procesamiento de tubería. Pipelining es una técnica en la que se superponen varias instrucciones durante la ejecución. La tubería se divide en etapas y estas etapas están conectadas entre sí para formar una estructura similar a una tubería. Las instrucciones entran por un extremo y salen por otro extremo.

9.6. Middleware en ASP.NET Core.

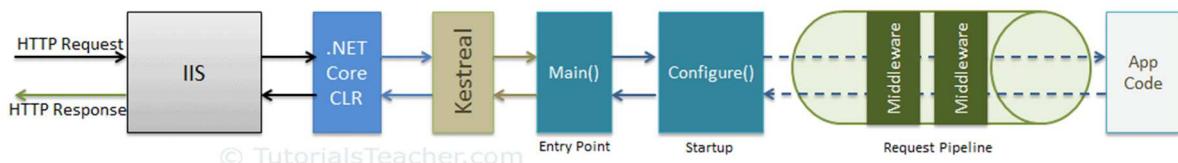
Ya que mencionamos la palabra Middleware, ASP.NET Core introdujo un nuevo concepto llamado Middleware. Un **middleware** no es más que un componente (clase) que se ejecuta en cada solicitud en la aplicación ASP.NET Core. En el ASP.NET clásico, HttpHandlers y HttpModules formaban parte de la canalización de solicitudes. El middleware es similar a HttpHandlers y HttpModules, donde ambos deben configurarse y ejecutarse en cada solicitud.

Por lo general, habrá varios middleware en la aplicación web ASP.NET Core. Puede ser un middleware proporcionado por el marco, agregado a través de NuGet o su propio middleware personalizado. Podemos

establecer el orden de ejecución del middleware en la tubería de solicitud. Cada middleware agrega o modifica la solicitud http y, opcionalmente, pasa el control al siguiente componente de middleware. La siguiente figura ilustra la ejecución de componentes de middleware.



Los middlewares crean la canalización de solicitudes. La siguiente figura ilustra el procesamiento de solicitudes de ASP.NET Core.



Middleware es un componente muy importante en cualquier aplicación de .NET Core, ya que permite:

- Captura excepciones síncronas y asíncronas de la canalización y genera respuestas de error HTML, así como tambien, sirve para detectar excepciones para registrarlas y volver a ejecutarlas en una canalización alternativa
- Proporcionar respuestas con códigos de estado entre 400 y 599.
- Muestre la página de bienvenida para la ruta raíz (FrontEnd, en MVC o Blazor).

La manera de crearlo en .NET Core es la siguiente: podemos configurar el middleware en el método **Configure** de la clase **Startup** usando la instancia **IApplicationBuilder**. El siguiente ejemplo agrega un único middleware usando el método **Run** que devuelve una cadena "¡Hola mundo!" en cada solicitud.

```

public class Startup
{
    public Startup()
    {
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory
loggerFactory)
    {
        //configure middleware using IApplicationBuilder here..
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
        // other code removed for clarity..
    }
}
  
```

Para en ASP.NET Core (Web API o MVC, o Blazor):

```
app.UseMiddleware<ExceptionHandlingMiddleware>();
```

En este caso, hay que implementar la clase **ExceptionHandlingMiddleware**, con la interfaz **IMiddleware**, como sigue:

```

internal sealed class ExceptionHandlingMiddleware : IMiddleware
{
    public ExceptionHandlingMiddleware()
    {
    }
    public Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        throw new NotImplementedException();
    }
}

```

Con esta implementación, ya podemos entonces canalizar el manejo de excepciones en todo el pipeline de ASP.NET Core, con el fin de tratar las excepciones que se generen durante su ejecución. Algunos componentes Nuget que realizan esta tarea para ASP.NET Core, son:

9.7. MediatR y FluentValidation: reemplazando la validación en ASP.NET Core.

Con todo esto que vimos anteriormente, ya podríamos armar nuestro Middleware y reforzar la fase de validación en el pipelining de ASP.NET Core con MediatR y FluentValidation, librerías que hemos incluido en nuestro proyecto. Para implementar la validación en nuestra canalización de CQRS, vamos a utilizar los conceptos de los que acabamos de hablar, y esos son: la clase **IPipelineBehavior** y FluentValidation de MediatR. Primero veamos la implementación de **ValidationBehavior**:

```

public class ValidationBehaviour<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>
    where TRequest : IRequest<TResponse>
{
    private readonly IEnumerable<IValidator<TRequest>> _validators;
    public ValidationBehaviour(IEnumerable<IValidator<TRequest>> validators) => _validators = validators;
    public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken, RequestHandlerDelegate<TResponse> next)
    {
        if (_validators.Any())
        {
            var context = new ValidationContext<TRequest>(request);
            var validationResults = await Task.WhenAll(_validators.Select(v => v.ValidateAsync(context, cancellationToken)));
            var failures = validationResults.SelectMany(r => r.Errors).Where(f => f != null)
                .GroupBy(x => x.PropertyName, x => x.ErrorMessage,
                    (propertyName, errorMessages) => new
                {
                    Key = propertyName,
                    Values = errorMessages.Distinct().ToArray()
                }).ToDictionary(x => x.Key, x => x.Values);

            if (failures.Count > 0)
                throw new ValidationException(failures);
        }
        return await next();
    }
}

```

Ahora, analicemos todo lo que está sucediendo en **ValidationBehavior**: en primer lugar, observamos cómo usamos la cláusula **where** para aplicar solo esta implementación de **IPipelineBehavior** para las clases que también implementan la interfaz **ICommand**. Esencialmente, solo permitimos que este **IPipelineBehavior** se ejecute si la solicitud que pasa por la canalización es un comando. Así es como lo logramos y reemplaza la validación clásica de ASP.NET Core.

A continuación, puede ver que estamos inyectando una colección de implementaciones de **IValidator** en el constructor. La biblioteca **FluentValidation** escaneará nuestro proyecto en busca de todas las implementaciones de **AbstractValidator** para un tipo determinado y luego nos proporcionará la instancia en tiempo de ejecución. Así es como podemos aplicar los validadores reales que implementamos en nuestro proyecto.

Por último, si hay algún error de validación, lanzamos una **ValidationException** que contiene el diccionario de errores de validación. Cuando se lanza la excepción debido a un error de validación, la canalización se cortocircuita y evitamos una mayor ejecución. Lo único que falta es manejar la excepción en algunas de las capas superiores de la aplicación y presentar una respuesta significativa al consumidor. Esto lo veremos más adelante cuando veamos el tema de manejo de excepciones en tiempo de ejecución en ASP.NET Core.

Tenemos una última cosa que hacer: registrar este Pipeline en el contenedor de servicios de ASP.NET Core, es decir, en **ConfigureServices** de **StartUp**:

```
services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehaviour<,>));
```

Dado que necesitamos validar todas y cada una de las solicitudes, las agregamos con el ciclo de vida del tipo **Transient** al contenedor.

9.8. Manejo de excepciones de validación.

Para manejar alguna excepción del tipo **ValidationException** que se genera cuando encontramos un error de validación, podemos usar la interfaz de **IMiddleware** de ASP.NET Core. Implementaremos un controlador global de excepciones:

```
public class ErrorHandlerMiddleware
{
    private readonly RequestDelegate _next;
    public ErrorHandlerMiddleware(RequestDelegate next) => _next = next;
    public async Task Invoke(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception error)
        {
            var response = context.Response;
            response.ContentType = "application/json";
            var responseModel = new ApiResponse<string>() { Succeeded = false, Message = error?.Message };

            switch (error)
            {
                case ValidateException e:
                    // custom application error
                    response.StatusCode = StatusCodes.Status422UnprocessableEntity;
                    responseModel.Errors = e.ErrorsDictionary;
                    break;

                default:
                    // unhandled error
                    response.StatusCode = StatusCodes.Status500InternalServerError;
                    break;
            }

            await response.WriteAsync(JsonConvert.SerializeObject(responseModel, new JsonSerializerSettings()
            {
                Culture = System.Globalization.CultureInfo.CurrentCulture,
                ReferenceLoopHandling = ReferenceLoopHandling.Ignore,
                NullValueHandling = NullValueHandling.Ignore,
                DateTimeZoneHandling = DateTimeZoneHandling.Local,
                FloatFormatHandling = FloatFormatHandling.DefaultValue,
                FloatParseHandling = FloatParseHandling.Decimal,
                ContractResolver = new CamelCasePropertyNamesContractResolver()
            }));
        }
    }
}
```

Y con esto, ya tenemos listo el Middleware para controlar este tipo de excepciones. Más adelante, cuando veamos como manejar globalmente las excepciones que puedan ocurrir en nuestro proyecto de Web API, ampliaremos esta clase para que controle los tipos de excepciones que puedan generarse. Finalmente, antes de poder ejecutar nuestra aplicación, debemos asegurarnos de haber registrado todos nuestros servicios con el contenedor de IoC:

1. Veamos como dar de alta nuestros Comandos y Consultas con MediatR. En la clase **Startup**, método **ConfigureServices** añadiremos las siguientes líneas:

```
services.AddMediatR(typeof(Application.AssemblyReference).Assembly);
services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehaviour<,>));
```

La primera llamada al método escaneará nuestro ensamblado de aplicación y agregará todos nuestros comandos, consultas y sus respectivos controladores al contenedor de IoC. La segunda llamada de método es nuestro paso de registro de **ValidationBehavior**. Sin esto, la canalización de validación no se ejecutaría en absoluto.

2. Luego, debemos asegurarnos de agregar también los validadores que implementamos con FluentValidation:

```
services.AddValidatorsFromAssembly(typeof(Application.AssemblyReference).Assembly);
```

3. Por último, debemos agregar `ExceptionHandlingMiddleware` en `ConfigureServices`:

```
services.AddTransient<ExceptionHandlingMiddleware>();
```

4. Y también en el método `Configure`:

```
app.UseMiddleware<ExceptionHandlingMiddleware>();
```

Eso es todo: hemos completado el registro de nuestros servicios con el contenedor de IoC. Ahora que tenemos todo conectado, podemos proceder a probar nuestra implementación.

9.9. Cambios a nuestro proyecto.

Para aplicar estos cambios requeridos, vamos entonces a hacer un cambio radical a nuestro proyecto justificando las razones de tal cambio:

- Hasta ahora, la validación fluida está en la capa de **Infraestructura**. Debería estar en la capa de **Application** para aplicar los patrones de diseño de Mediator y CQRS. Esa capa, no la tenemos realmente.
- La capa de Infraestructura está todo en un solo lugar. Tenemos persistencia de datos, Unit Of Work y la parte de los repositorios y servicios en un solo proyecto. **Tenemos que separarlos**.
- La capa de Presentación que es la API, está creciendo mucho la parte de la **IoC (Inversion Of Control)**. Tenemos que simplificarlo. Cuando ocurre una excepción, **se muestra el detalle completo de la excepción**, lo cual, es información valiosa que un atacante puede aprovechar para atacar a nuestra aplicación Web API. Tenemos que proteger este aspecto, pero nuestro proyecto no tiene el Middleware configurado.

Para resolver estos aspectos, vamos a presentar la estructura final de nuestro proyecto con los cambios antes mencionados:

Capa nueva	Detalle	Nombre del proyecto
Domain (antes Core)	Definición de los casos de uso, entidades de Base de Datos, los tipos de excepción y features adicionales.	CA.Domain
Infrastructure.Persistence	Capa donde se guardan los contextos de Base de Datos de manera separada.	CA.Infrastructure.Persistence
Infrastructure.UnitOfWork	Capa donde se implementa Unit Of Work para manejar multiples instancias de Bases de Datos, y la implementación de los patrones de diseño DbFactory, Repository Base.	CA.Infrastructure.UnitOfWork
Infrastructure.Common	Capa donde se aplican las implementaciones de Repository y Services de las entidades de Base de Datos.	CA.Infrastructure.Common
Application	Capa donde se definen los patrones de diseño CQRS, Mediator y los behaviours para validación, logging, autenticación y otras funcionalidades, con el fin de gestionar y controlar el pipeline de ASP.NET Core.	CA.Application
Presentation	Capa donde se muestra los controller de la Web API y rediseño de IoC y definición del Middleware de la aplicación en general.	CA.Api
Cada una de las capas tendrá su propio IoC, excepto la capa de Domain.		

Con esto, respetamos fielmente la implementación del modelo de Clean Architecture. Para cada proyecto, usaremos las librerías correspondientes:

Capa nueva	Detalle	Nombre del proyecto
Domain (antes Core)	<ul style="list-style-type: none"> • MediatR • FluentValidation • Microsoft.EntityFrameworkCore 	CA.Domain
Infrastructure.Persistence	<ul style="list-style-type: none"> • Microsoft.EntityFrameworkCore • Microsoft.EntityFrameworkCore.Tools • Microsoft.EntityFrameworkCore.Design • Microsoft.EntityFrameworkCore.SqlServer • Microsoft.EntityFrameworkCore.Relational • Microsoft.EntityFrameworkCore.Abstractions 	CA.Infrastructure.Persistence
Infrastructure.UnitOfWork	<ul style="list-style-type: none"> • Microsoft.EntityFrameworkCore 	CA.Infrastructure.UnitOfWork
Infrastructure.Common	<ul style="list-style-type: none"> • Ardalis.GuardClauses • Microsoft.AspNetCore.Mvc • Microsoft.AspNetCore.Mvc.NewtonsoftJson • Microsoft.EntityFrameworkCore • Microsoft.EntityFrameworkCore.Tools • Microsoft.EntityFrameworkCore.Design • Microsoft.EntityFrameworkCore.SqlServer • Microsoft.EntityFrameworkCore.Relational • Microsoft.EntityFrameworkCore.Abstractions • FluentValidation • FluentValidation.DependencyInjectionExtensions • AutoMapper.Extensions.Microsoft.DependencyInjection 	CA.Infrastructure.Common
Application	<ul style="list-style-type: none"> • AutoMapper • AutoMapper.Extensions.Microsoft.DependencyInjection • FluentValidation • FluentValidation.DependencyInjectionExtensions • Microsoft.EntityFrameworkCore.InMemory • MediatR.Extensions.Microsoft.DependencyInjection • Microsoft.EntityFrameworkCore • System.Text.Json 	CA.Application
Presentation	<ul style="list-style-type: none"> • MediatR • MediatR.Extensions.Microsoft.DependencyInjection • Microsoft.AspNetCore.Mvc.NewtonsoftJson • Microsoft.EntityFrameworkCore.Design 	CA.Api

No haremos más cambios al proyecto que tenemos hasta ahora, después de esta nueva reorganización. Excepto la capa de Domain, todas las demás capas, tendrán su propio IoC con el fin de no extender el IoC de la capa de Presentación, que es obviamente, nuestra Web API.

9.10. Aplicando a nuestro código.

Vamos entonces a empezar a aplicar lo anteriormente explicado a nuestro proyecto. Asumiendo que hemos hecho los ajustes antes mencionados, vamos a realizar parte por parte:

Capa de Dominio (CA.Domain). En la capa de dominio, hacemos lo siguiente:

1. Hay que incluir las librerías requeridas para esta capa, explicado en la tabla anterior, de la sección 9.9.
2. En la carpeta **CA.Domain\Wrapper**, revisar que exista el archivo **ApiResponse.cs** y asegurarse que tenga la siguiente estructura:

```

using System.Collections.Generic;

namespace CA.Domain.Wrappers
{
    public class ApiResponse<T>
    {
        public ApiResponse() { }
        public ApiResponse(T data, string message = null)
        { Succeeded = true; Message = message; Data = data; }
        public ApiResponse(string message, bool isOK = false)
        { Succeeded = isOK; Message = message; }
        public bool Succeeded { get; set; }
        public string Message { get; set; }
        public IReadOnlyDictionary<string, string[]> Errors { get; set; }
        public T Data { get; set; }
    }
}

```

3. En la carpeta **Exception**, eliminemos la carpeta de **Entities** puesto que originalmente hicimos esa carpeta en el capítulo anterior para cada una de las entidades y con el fin de reorganizar nuestro proyecto,

usaremos finalmente las excepciones de las entidades, con las clases del tipo **Exception** encontradas en la carpeta **Core\Entities**, incluida en esta carpeta.

4. En la carpeta **CA.Domain\Exceptions**, crearemos una carpeta llamada **Api** y dentro de ella un archivo de clase llamado **ApiException.cs**, el cual, manejará las excepciones que se puedan generar en tiempo de ejecución para las operaciones de la WebApi en general. Su estructura es la siguiente:

```
public class ApiException : Exception
{
    public ApiException() : base() { }
    public ApiException(string message) : base(message) { }
    public ApiException(string message, params object[] args)
        : base(string.Format(CultureInfo.CurrentCulture, message, args)) { }
}
```

5. En la carpeta **CA.Domain\Exceptions\Core\Entities**, crearemos un archivo de clase llamado **EntityAlreadyExistsException.cs**, cuyo objetivo es lanzar una excepción indicando que si una entidad existe en el contexto de datos, al realizarse una consulta en Entity Framework. Su estructura es la siguiente:

```
public class EntityAlreadyExistException : Exception
{
    public Type EntityType { get; set; }
    public object ValueInfo { get; set; }
    public EntityAlreadyExistException() : base() { }
    public EntityAlreadyExistException(Type entityType) : this(entityType, null, null) { }
    public EntityAlreadyExistException(Type entityType, object valueInfo) : this(entityType, valueInfo, null) { }
    public EntityAlreadyExistException(Type entityType, object valueInfo, Exception innerException) : base(
        valueInfo == null ? $"An entity already exists. Entity type: '{entityType.FullName}'." :
        $"An entity with the value '{valueInfo}' already exists. Entity type: '{entityType.FullName}'.",
        innerException)
    {
        EntityType = entityType;
        ValueInfo = valueInfo;
    }
    public EntityAlreadyExistException(string message) : base(message) { }
    public EntityAlreadyExistException(string message, Exception innerException) : base(message, innerException) { }
}
```

Guardemos los cambios y dejemos las demás carpetas como estaban.

6. Creamos una carpeta nueva llamada **Validation**, dentro de la carpeta **CA.Domain\Exceptions\Core**, llamado **ValidateException.cs**. Su finalidad es generar excepciones para FluentValidation y su estructura es la siguiente:

```
public class ValidateException : Exception
{
    public IReadOnlyDictionary<string, string[]> ErrorsDictionary { get; }
    public ValidateException(IReadOnlyDictionary<string, string[]> errorDictionary) :
        base("One or more validation errors occurred") => ErrorsDictionary = errorDictionary;
}
```

7. En la carpeta llamada **DTO**, hay que crear una carpeta llamada **Base** y dentro de ella, un archivo de clase llamado **CommandDTO.cs**. Esta clase es una clase abstracta para que los DTOs que tengan iniciales de Create, Delete y Update, hereden de esta clase y puedan ser reconocidos como del tipo **CommandDTO**. Su estructura es la siguiente:

```
namespace CA.Domain.DTO.Base
{
    public abstract class CommandDTO { }
}
```

8. En las carpetas de los objetos DTO, aplicar la herencia de la clase abstracta antes creada. Por ejemplo, en la carpeta **ArticleDTO**, revisemos el archivo **CreateArticleDTO.cs**. Decoremos la línea de código de esa clase como sigue:

```

using MediatR;
using CA.Domain.Wrappers;
using CA.Domain.DTO.Base;

namespace CA.Domain.DTO
{
    public class CreateArticleDTO : CommandDTO, IRequest<ApiResponse<CreateArticleDTO>>
    {
        public string ShortName { get; set; }
        public string Description { get; set; }
        public int DepartamentId { get; set; }
    }
}

```

Lo que estamos haciendo es cambiar el objeto `CreateArticleDTO` en un `Command` para MediatR y que herede de la interfaz `IRequest<ApiResponse<CreateArticleDTO>>` para que `IRequest` genere el response del tipo `ApiResponse<T>`. `T` debe ser el mismo objeto DTO de operación de comando, es decir, `CreateArticleDTO`. Guardemos los cambios y repetimos el mismo procedimiento para los demás archivos: `DeleteArticleDTO.cs` y `UpdateArticleDTO.cs` y guardemos los cambios otra vez. Aplicar este paso a los demás DTOS que contengan esas clases con las iniciales de las operaciones de comando (**Create, Delete y Update**) antes mencionados y guarde los cambios de nuevo.

9. El archivo `RegexExtensions.cs` del proyecto `CA.Infrastructure`, en la carpeta `CA.Infrastructure\Extensions\Base` a una nueva carpeta llamada `CA.Domain\Features`. Eliminemos las demás carpetas contenidas de la carpeta de origen y cambiemos el namespace del archivo copiado en `CA.Domain\Features` como namespace `CA.Domain.Features`. Guardemos cambios.
10. Ahora, en la carpeta `CA.Domain\Interfaces\Services\Base`, hay dos archivos: `ICRUDService.cs` y `IRService.cs`. Anteriormente, la definición de `ICRUDService.cs` era así:

```
public interface ICRUDService<TGetDto, TAddDto, TUpdDto, TDelDto, TKey, TEntity, TRepoAll, TContext,
```

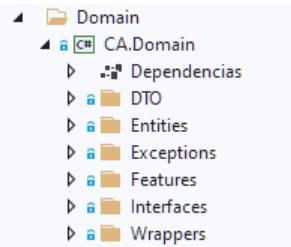
Reemplazamos esa línea como sigue, pulsando la combinación de teclas **Ctrl + RR**, reemplazamos las variables `TAddDto`, `TUpdDto` y `TDelDto` por `TCommandDTO` y `TGetDto` como `TQueryDTO`. Al hacer ese cambio, debe quedar algo así:

```
1 public interface ICRUDService<TQueryDTO, TCommandDTO, TKey, TEntity, TRepoAll, TContext>
```

En `IRService.cs` debe quedar algo así:

```
public interface IRService<TQueryDTO, TKey, TEntity, TRepoRead, TContext>
```

Guardemos los cambios y compilemos el proyecto. No debería generar algún error de compilación. Nuestro proyecto de Domain debe quedar algo como esto:



Capa de Persistencia de Datos (CA.Infrastructure.Persistence). En la capa de persistencia de datos, hacemos lo siguiente:

1. Crear un nuevo proyecto, del tipo de librería llamado `CA.Infrastructure.Persistence`.
2. Hay que incluir las librerías requeridas para esta capa, explicado en la tabla anterior, de la sección 9.9.
3. Las carpetas incluidas de la carpeta llamada `CA.Infrastructure\Persistence\Data`, pasárlas al nuevo proyecto anteriormente generado y cambiar el espacio de nombres correspondientes a cada archivo y

deben empezar como namespace **CA.Infrastructure.Persistence**. Eliminar esa carpeta antes mencionada.

4. Agregar la referencia de proyecto de **CA.Domain** y guardar los cambios.
5. En el proyecto nuevo generado, crear una carpeta llamada **ServiceCollection** y dentro de ella un archivo llamado **ServiceExtension.cs**, cuyo contenido es el siguiente:

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

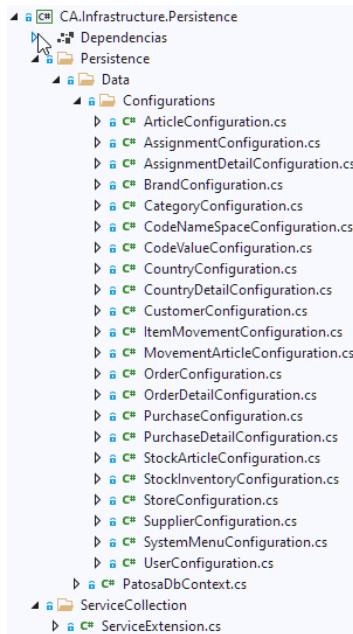
using CA.Infrastructure.Persistence.Data;

namespace CA.Infrastructure.Persistence.ServiceCollection
{
    public static class ServiceExtension
    {
        public static void AddPersistenceLayer(this IServiceCollection services, IConfiguration configuration)
        {
            /* Contextos de Bases de Datos. */
            services.AddDbContext<PatosaDbContext>(options => { options.UseSqlServer(configuration.GetConnectionString("PatosaDbContext")); });

            /* DbFactory pattern. */
            /* Agregar aquí las implementaciones de Factory Pattern, asociadas a cada contexto de Base de Datos... */
            services.AddScoped<Func<PatosaDbContext>>((provider) => () => provider.GetService<PatosaDbContext>());
        }
    }
}
```

Creamos un método estático llamado **AddPersistenceLayer** y **ServiceExtension** como clase estática. El método antes mencionado acepta como parámetros **IServiceCollection** y **IConfiguration**, cuya definición, lo explicamos en el capítulo 7. De manera que con este cambio, ya podemos definir los contextos de Base de Datos y su implementación por el patrón de diseño Factory.

6. Guardemos los cambios y compilemos. Ya separamos la persistencia de datos en un proyecto diferente. Su estructura nueva quedaría así:



Capa de UnitOfWork (CA.Infrastructure.UnitOfWork). En la capa de Unit Of Work, hacemos lo siguiente:

1. Crear un nuevo proyecto de librería de clase llamado **CA.Infrastructure.UnitOfWork**.
2. Hay que incluir las librerías requeridas para esta capa, explicado en la tabla anterior, de la sección 9.9.
3. Del proyecto anterior, **CA.Infrastructure**, movemos la carpeta **CA.Infrastructure\Persistence\Base** al nuevo proyecto y debe quedar como **CA.Infrastructure.UnitOfWork\Base**. En esa carpeta, cambiar la

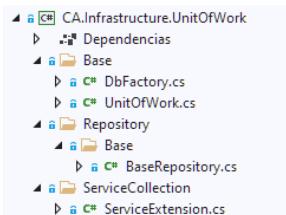
definición de los namespace que empiecen como namespace **CA.Infrastructure.UnitOfWork** y guardemos los cambios.

4. Ahora movamos la carpeta **CA.Infrastructure\Persistence\Repository\Base** a una nueva carpeta llamada **CA.Infrastructure.UnitOfWork\Repository\Base** con todo y sus archivos incluidos. Cambiar el namespace como namespace **CA.Infrastructure.UnitOfWork**.
5. Agregar las referencias del proyecto **CA.Domain** y **CA.Infrastructure.Persistence** al proyecto nuevo.
6. En el proyecto nuevo generado, crear una carpeta llamada **ServiceCollection** y dentro de ella un archivo llamado **ServiceExtension.cs**, cuyo contenido es el siguiente:

```
public static class ServiceExtension
{
    public static void AddUnitOfWorkLayer(this IServiceCollection services)
    {
        /* Factory y Unit Of Work. */
        services.AddScoped<IDbFactory<PatosaDbContext>, DbFactory<PatosaDbContext>>();
        services.AddScoped<IUnitOfWork<PatosaDbContext>, UnitOfWork<PatosaDbContext>>();
    }
}
```

Creamos un método estático llamado **AddUnitOfWorkLayer** y **ServiceExtension** como clase estática. El método antes mencionado acepta como parametros **IServiceCollection**. De manera que con este cambio, ya podemos definir las implementaciones **IDbFactory** y **IUnitOfWork** para cada instancia o contexto de Base de Datos.

7. Guardemos los cambios y compilemos. Nuestro proyecto de **UnitOfWork** ya está separado en una capa aparte y quedaría así:



Capa de elementos comunes de Infraestructura (CA.Infrastructure.Common). En la capa común de Infraestructura, hacemos lo siguiente:

1. Crear un nuevo proyecto de librería de clase llamado **CA.Infrastructure.Common**.
2. Hay que que incluir las librerías requeridas para esta capa, explicado en la tabla anterior, de la sección 9.9.
3. Del proyecto anterior, **CA.Infrastructure**, movemos la carpeta **CA.Infrastructure\Persistence\Repository** al nuevo proyecto y debe quedar como **CA.Infrastructure.Common\Repositories**. En esa carpeta, cambiar la definición de los namespace que empiecen como namespace **CA.Infrastructure.Common.Repositories** y guardemos los cambios.
4. Repitamos el mismo paso moviendo la carpeta **CA.Infrastructure\Persistence\Services** a **CA.Infrastructure.Common\Services** y cambiemos en cada uno de los archivos, el namespace de cada archivo y que empiecen como namespace **CA.Infrastructure.Common.Services**. Guardemos los cambios.
5. Ahora, en la carpeta **CA.Infrastructure.Common\Repositories**, vayamos a cambiar cada uno de los archivos de repositorio. Solo hay que asegurarse que tengan el namespace correcto. Elimine la carpeta **Base**, puesto que ya tenemos una clase llamada **BaseRepository** desde la capa de **CA.Infrastructure.UnitOfWork**.
6. Ahora, en la carpeta **CA.Infrastructure.Common\Services\Base**, hay dos archivos: **CRUDService.cs** y **RService.cs**. Se tienen que cambiar la decoración de la línea de la clase como sigue:

```
public abstract class CRUDService<TQueryDTO, TCommandDTO, TKey, TEntity, TRepoAll, TContext> : ICRUDService<TQueryDTO, TCommandDTO, TKey, TEntity, TRepoAll, TContext>
{
    where TEntity : class, IEntityBase<TKey>
    where TRepoAll : IBaseRepository<
```

Esto lo hacemos ya heredando de las interfaces **ICRUDService** y **IRService** para que se tomen los tipos genéricos **TQueryDTO** y **TCommandDTO**. Guardemos los cambios y apliquemos el mismo procedimiento a **RService.cs**.

```
public abstract class RService<TQueryDTO, TKey, TEntity, TRepoRead, TContext> :  
    IRServicet<TQueryDTO, TKey, TEntity, TRepoRead, TContext>  
    where TEntity : class, IEntityBase<TKey>  
    where TRepoRead : IReadRepository<, TContext>  
    where TContext : DbContext, new()
```

7. Guardemos los cambios. Ahora tenemos que ajustar los archivos contenidos en la carpeta de Services. Por ejemplo, en la carpeta **ArticleService.cs**, el decorado de la clase sería así:

```
public class ArticleService : CRUDService<ArticleDTO, CommandDTO, int,  
    Article, IArticleRepository<PatosaDbContext>,  
    PatosaDbContext>, IArticleService
```

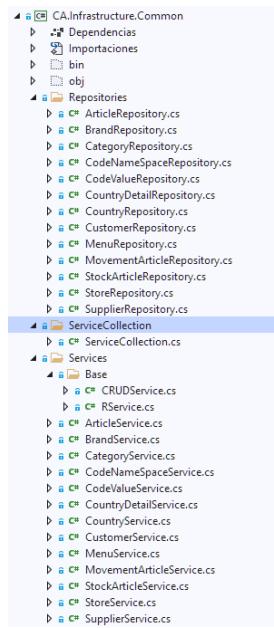
Con la combinación de teclas **Ctrl + RR**, ajustemos las variables de los tipos genéricos de cada una de las funciones del servicio. Guardemos los cambios y apliquemos esto mismo a los demás archivos de servicios.

8. Agregar las referencias del proyecto **CA.Domain**, **CA.Infrastructure.Persistence** y **CA.Infrastructure.UnitOfWork** al proyecto nuevo.
9. En el proyecto nuevo generado, crear una carpeta llamada **ServiceCollection** y dentro de ella un archivo llamado **ServiceExtension.cs**, cuyo contenido es el siguiente:

```
public static void AddCommonLayer(this IServiceCollection services)  
{  
    /* Repositorios */  
    services.AddTransient<IArticleRepository<PatosaDbContext>, ArticleRepository>();  
    services.AddTransient<IStoreRepository<PatosaDbContext>, StoreRepository>();  
    services.AddTransient<IBrandRepository<PatosaDbContext>, BrandRepository>();  
    services.AddTransient<ISupplierRepository<PatosaDbContext>, SupplierRepository>();  
    services.AddTransient<ICategoryRepository<PatosaDbContext>, CategoryRepository>();  
    services.AddTransient<IMenuRepository<PatosaDbContext>, MenuRepository>();  
    services.AddTransient<ICodeValueRepository<PatosaDbContext>, CodeValueRepository>();  
    services.AddTransient<ICodeNameSpaceRepository<PatosaDbContext>, CodeNameSpaceRepository>();  
    services.AddTransient<IMovementArticleRepository<PatosaDbContext>, MovementArticleRepository>();  
    services.AddTransient<IStockArticleRepository<PatosaDbContext>, StockArticleRepository>();  
    services.AddTransient<ICustomerRepository<PatosaDbContext>, CustomerRepository>();  
    services.AddTransient<ICountryRepository<PatosaDbContext>, CountryRepository>();  
    services.AddTransient<ICountryDetailRepository<PatosaDbContext>, CountryDetailRepository>();  
  
    /* Servicios */  
    services.AddTransient<IArticleService, ArticleService>();  
    services.AddTransient<IStoreService, StoreService>();  
    services.AddTransient<IBrandService, BrandService>();  
    services.AddTransient<ISupplierService, SupplierService>();  
    services.AddTransient<ICategoryService, CategoryService>();  
    services.AddTransient<IMenuService, MenuService>();  
    services.AddTransient<ICodeValueService, CodeValueService>();  
    services.AddTransient<ICodeNameSpaceService, CodeNameSpaceService>();  
    services.AddTransient<IMovementArticleService, MovementArticleService>();  
    services.AddTransient<IStockArticleService, StockArticleService>();  
    services.AddTransient<ICustomerService, CustomerService>();  
    services.AddTransient<ICountryService, CountryService>();  
    services.AddTransient<ICountryDetailsService, CountryDetailService>();  
}
```

Creamos un método estático llamado **AddCommonLayer** y **ServiceExtension** como clase estática. El método antes mencionado acepta como parámetros **IServiceCollection**. De manera que con este cambio, ya podemos definir las implementaciones de los servicios y repositorios para cada instancia o contexto de Base de Datos.

10. Guardemos los cambios y compilemos. Nuestro proyecto nuevo quedaría como sigue: los repositorios y servicios ya están separados en un proyecto aparte.



Capa de Aplicación (CA.Application). En la capa de aplicación, hacemos lo siguiente:

1. Crear un nuevo proyecto de librería de clase llamado **CA.Application**.
2. Hay que incluir las librerías requeridas para esta capa, explicado en la tabla anterior, de la sección 9.9.
3. Del proyecto anterior, **CA.Infrastructure**, movemos la carpeta **CA.Infrastructure\Mappers** y **CA.Infrastructure\Validator** al nuevo proyecto y debe quedar como **CA.Application\Mappers** y **CA.Application\Validators** respectivamente. En esa carpeta, cambiar la definición de los namespace que empiecen como namespace **CA.Application.Mappers** y **CA.Application.Validators** guardemos los cambios.
4. Creamos otras carpetas nuevas: **Behaviours**, **Querys** y **Handlers**.
5. En la carpeta **Behaviours**, hay que crear un archivo llamado **ValidationBehaviour.cs**, cuyo contenido es el siguiente:

```
public class ValidationBehaviour<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>
{
    where TRequest : IRequest<TResponse>
    {
        private readonly IEnumerable<IValidator<TRequest>> validators;
        public ValidationBehaviour(IEnumerable<IValidator<TRequest>> validators) => _validators = validators;
        public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken, RequestHandlerDelegate<TResponse> next)
        {
            if (_validators.Any())
            {
                var context = new ValidationContext<TRequest>(request);
                var validationResults = await Task.WhenAll(_validators.Select(v => v.ValidateAsync(context, cancellationToken)));
                var failures = validationResults.SelectMany(r => r.Errors).Where(f => f != null)
                    .GroupBy(x => x.PropertyName, x => x.ErrorMessage,
                        (propertyName, errorMessages) => new
                        {
                            Key = propertyName,
                            Values = errorMessages.Distinct().ToArray()
                        }).ToDictionary(x => x.Key, x => x.Values);

                if (failures.Count > 0)
                    throw new ValidateException(failures);
            }
            return await next();
        }
    }
}
```

6. Guardemos los cambios y vayamos a la carpeta de **Querys**. El archivo que vamos a crear se llama **GetArticleQuery.cs** y su estructura es la siguiente:

```

using System.Collections.Generic;
using MediatR;
using CA.Domain.DTO;
namespace CA.Application.Queries
{
    public class GetAllArticleQuery : IRequest<IEnumerable<ArticleDTO>> { }
    public class GetArticleQuery : IRequest<ArticleDTO>
    {
        public int Id { get; }
        public GetArticleQuery(int id) => Id = id;
    }
}

```

Vemos que estamos creando dos clases: **GetAllArticleQuery** y **GetArticleQuery**. Ambas clases heredan de la interface **IRequest<T>** de MediatR. En este caso, para la primera clase sería un **IRequest<IEnumerable<ArticleDTO>>** y para la segunda **IRequest<ArticleDTO>** con el fin de separar los request para consultar un artículo y todos los artículos respectivamente. Guardemos los cambios y apliquemos esto para los demás DTOS.

7. En la carpeta de **Handlers**, hay que crear dos carpetas: **Command** y **Query**.
8. En la carpeta **CA.Application\Handlers\Query**, hay que crear dos carpetas más: **All** y **Single**.
9. En la carpeta **CA.Application\Handlers\Command**, hay que crear tres carpetas más: **Create**, **Update** y **Delete**.
10. En la carpeta **CA.Application\Handlers\Query\Single**, creamos un archivo llamado **GetArticleHandler.cs**. Su estructura es la siguiente:

```

using System.Threading;
using System.Threading.Tasks;
using MediatR;

using CA.Domain.DTO;
using CA.Application.Queries;
using CA.Domain.Interfaces.Services;

namespace CA.Application.Handlers.Query
{
    public class GetArticleHandler : IRequestHandler<GetArticleQuery, ArticleDTO>
    {
        private readonly IArticleService _articleService;
        public GetArticleHandler(IArticleService articleService) => _articleService = articleService;
        public async Task<ArticleDTO> Handle(GetArticleQuery request, CancellationToken cancellationToken) =>
            await _articleService.FindArticleAsync(request.Id, cancellationToken);
    }
}

```

Notemos que estamos usando el servicio de Artículos, creado desde **CA.Infrastructure.Common**. Hay que inyectarlo y en la función **Handle**, tenemos que indicarle que llame a la función **FindArticleAsync** del servicio de manera asíncrono.

11. Guardemos los cambios y ahora vayamos a la carpeta **CA.Application\Handlers\Query\All** y creamos un archivo llamado **GetAllArticleHandler.cs**. Su estructura es la siguiente:

```

using System.Threading;
using System.Threading.Tasks;
using System.Collections.Generic;
using MediatR;

using CA.Domain.DTO;
using CA.Application.Queries;
using CA.Domain.Interfaces.Services;

namespace CA.Application.Handlers.Query
{
    public class GetAllArticleHandler : IRequestHandler<GetAllArticleQuery, IEnumerable<ArticleDTO>>
    {
        private readonly IArticleService _articleService;
        public GetAllArticleHandler(IArticleService articleService) => _articleService = articleService;
        public async Task<IEnumerable<ArticleDTO>> Handle(GetAllArticleQuery request, CancellationToken cancellationToken) =>
            await _articleService.GetArticlesAsync(cancellationToken);
    }
}

```

La estructura de este archivo es similar que en el archivo del paso 10.

12. Repitamos los pasos 10 y 11 para los demás DTO.
13. Guardemos los cambios y vayamos a la carpeta **CA.Application\Handlers\Command\Create**. Crearemos un archivo llamado **CreateArticleHandler.cs**. Su estructura es la siguiente:

```

using System.Threading;
using System.Threading.Tasks;

using MediatR;

using CA.Domain.DTO;
using CA.Domain.Wrappers;
using CA.Domain.Interfaces.Services;

namespace CA.Application.Handlers.Command
{
    public class CreateArticleHandler : IRequestHandler<CreateArticleDTO, ApiResponse<CreateArticleDTO>>
    {
        private readonly IArticleService _articleService;
        public CreateArticleHandler(IArticleService articleService) => _articleService = articleService;
        public async Task<ApiResponse<CreateArticleDTO>> Handle(CreateArticleDTO request, CancellationToken cancellationToken)
        {
            var entity = await _articleService.InsertArticleAsync(request, cancellationToken);
            return new ApiResponse<CreateArticleDTO>(entity, $"The article with name '{entity.ShortName}' was created successfully.");
        }
    }
}

```

Aquí cambia el contexto de la clase: como este handler es del tipo **ICommand**, se tiene que inyectar a **IRequestHandler** los tipos **CreateArticleDTO** y **ApiResponse<CreateArticleDTO>**, puesto que se envía un **CreateArticleDTO** y se devuelve un **ApiResponse<CreateArticleDTO>**, si la operación fue satisfactoria. Se inyecta **IArticleService** a la clase y se invoca el método de agregar un nuevo registro, que es en este caso **InsertArticleAsync**.

14. En la carpeta **CA.Application\Handlers\Command\Update**, creamos un archivo llamado **UpdateArticleHandler.cs** y su estructura es similar:

```

using System.Threading;
using System.Threading.Tasks;

using MediatR;

using CA.Domain.DTO;
using CA.Domain.Wrappers;
using CA.Domain.Interfaces.Services;

namespace CA.Application.Handlers.Command
{
    public class UpdateArticleHandler : IRequestHandler<UpdateArticleDTO, ApiResponse<UpdateArticleDTO>>
    {
        private readonly IArticleService _articleService;
        public UpdateArticleHandler(IArticleService articleService) => _articleService = articleService;
        public async Task<ApiResponse<UpdateArticleDTO>> Handle(UpdateArticleDTO request, CancellationToken cancellationToken)
        {
            var entity = await _articleService.UpdateArticleAsync(request, cancellationToken);
            return new ApiResponse<UpdateArticleDTO>(entity, $"The article with Id {entity.Id} was successfully updated.");
        }
    }
}

```

15. Y finalmente, en la carpeta **CA.Application\Handlers\Command\Delete**, creamos un archivo llamado **DeleteArticleHandler.cs** y su estructura es similar.

```

using System.Threading;
using System.Threading.Tasks;

using MediatR;

using CA.Domain.DTO;
using CA.Domain.Wrappers;
using CA.Domain.Interfaces.Services;

namespace CA.Application.Handlers.Command
{
    public class DeleteArticleHandler : IRequestHandler<DeleteArticleDTO, ApiResponse<DeleteArticleDTO>>
    {
        private readonly IArticleService _articleService;
        public DeleteArticleHandler(IArticleService articleService) => _articleService = articleService;
        public async Task<ApiResponse<DeleteArticleDTO>> Handle(DeleteArticleDTO request, CancellationToken cancellationToken)
        {
            var entity = await _articleService.DeleteArticleAsync(request, request.AutoSave, cancellationToken);
            return new ApiResponse<DeleteArticleDTO>(entity, $"The article with Id {entity.Id} was successfully deleted.");
        }
    }
}

```

16. Repetir los pasos 13, 14 y 15 para los demás objetos DTO que puedan realizar las operaciones de **Create**, **Delete** y **Update**.
17. Guardemos los cambios y en el archivo **AutoMapperProfile.cs** de la carpeta **CA.Application\Mappings** hagamos el siguiente ajuste para los objetos DTO relacionados con los artículos:

```

using AutoMapper;
using CA.Domain.DTO;
using CA.Domain.Entities;
namespace CA.Application.Mappings
{
    public class AutoMapperProfile : Profile
    {
        public AutoMapperProfile()
        {
            /* Articulos */
            CreateMap<Article, ArticleDTO>().ReverseMap();
            CreateMap<CreateArticleDTO, Article>().ReverseMap();
            CreateMap<CreateArticleDTO, ArticleDTO>().ReverseMap();
            CreateMap<UpdateArticleDTO, Article>().ReverseMap();
            CreateMap<UpdateArticleDTO, ArticleDTO>().ReverseMap();
            CreateMap<DeleteArticleDTO, Article>().ReverseMap();
            CreateMap<DeleteArticleDTO, ArticleDTO>().ReverseMap();

            /* Los demás DTO's... */
        }
    }
}

```

Notemos que se tienen que agregar nuevas líneas de mapeo de objetos entre los DTO's de comandos a DTO de consulta. En este caso, los DTO de comando serían `CreateArticleDTO`, `UpdateArticleDTO` y `DeleteArticleDTO` a `ArticleDTO`, para evitar un error de mapeo en tiempo de ejecución. Apliquemos este principio a los demás DTO que tenemos en el proyecto.

18. Guardemos los cambios. Finalmente revisemos la carpeta **CA.Application\Validators**. Ajustemos los espacios de nombres de cada uno de los archivos a **CA.Application.Validators** y guardemos los cambios aplicados.
19. Agregar las referencias del proyecto **CA.Domain**, **CA.Infrastructure.Persistence**, **CA.Infrastructure.UnitOfWork** y **CA.Infrastructure.Common** al proyecto nuevo.
20. En el proyecto nuevo generado, crear una carpeta llamada **ServiceCollection** y dentro de ella un archivo llamado **ServiceExtension.cs**, cuyo contenido es el siguiente:

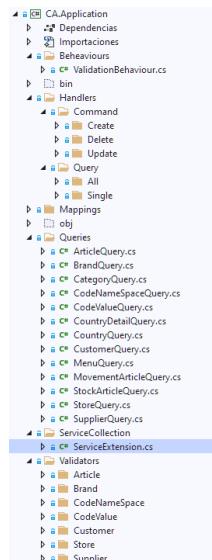
```

public static void AddApplicationLayer(this IServiceCollection services)
{
    services.AddAutoMapper(Assembly.GetExecutingAssembly());
    services.AddValidatorsFromAssembly(Assembly.GetExecutingAssembly());
    services.AddMediatR(Assembly.GetExecutingAssembly());
    services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehaviour<,>));
}

```

Creamos un método estático llamado **AddApplicationLayer** y **ServiceExtension** como clase estática. El método antes mencionado acepta como parámetro **IServiceCollection**. De manera que con este cambio, ya podemos definir las implementaciones de FluentValidation, Automapper, MediatR y el pipeline de validación de DTO.

21. Guardemos los cambios y compilemos. La estructura del proyecto **CA.Application** ya está lista:



22. Finalmente eliminamos el proyecto anterior de **CA.Infrastructure** que teníamos anteriormente y compilamos finalmente todo. Habrá errores en el proyecto **CA.Api**, pero eso a continuación los corregimos.

Capa de Presentación (CA.Presentation). En la capa de presentación, hacemos lo siguiente:

1. Hay que incluir las librerías requeridas para esta capa, explicado en la tabla anterior, de la sección 9.9.
2. Creamos dos nuevas carpetas: **Middleware** y **ServiceCollection**.
3. En la carpeta de **CA.Api\Middleware**, creamos un archivo llamado **ErrorHandleMiddleware.cs**, cuya finalidad es el control de excepciones durante la ejecución de la WebApi y su estructura es la siguiente:

```
public class ErrorHandleMiddleware : IMiddleware
{
    public ErrorHandleMiddleware() { }
    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        try
        {
            await next(context);
        }
        catch (Exception error)
        {
            var response = context.Response;
            response.ContentType = "application/json";
            var responseModel = new ApiResponse<string>() { Succeeded = false, Message = error?.Message };

            switch (error)
            {
                case ApiException e:
                    // custom application error
                    response.StatusCode = StatusCodes.Status400BadRequest;
                    break;

                case ValidateException e:
                    // custom application error
                    response.StatusCode = StatusCodes.Status422UnprocessableEntity;
                    responseModel.Errors = e.ErrorsDictionary;
                    break;

                default:
                    // unhandled error
                    response.StatusCode = StatusCodes.Status500InternalServerError;
                    break;
            }

            await response.WriteAsync(JsonConvert.SerializeObject(responseModel, new JsonSerializerSettings()
            {
                Culture = System.Globalization.CultureInfo.CurrentCulture,
                ReferenceLoopHandling = ReferenceLoopHandling.Ignore,
                NullValueHandling = NullValueHandling.Ignore,
                DateTimeZoneHandling = DateTimeZoneHandling.Local,
                FloatFormatHandling = FloatFormatHandling.DefaultValue,
                FloatParseHandling = FloatParseHandling.Decimal,
                ContractResolver = new CamelCasePropertyNamesContractResolver()
            }));
        }
    }
}
```

4. En la carpeta **CA.Api\ServiceCollection**, creamos dos archivos: el primero se llama **AppBuilderExtension.cs** y tenemos que separar la función **Configure** con el siguiente código:

```
public static class AppBuilderExtension
{
    public static void InitConfigurationAPI(this IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
            app.UseDeveloperExceptionPage();

        app.UseHttpsRedirection();
        app.UseRouting();
        app.UseAuthorization();
        app.UseMiddleware<ErrorHandleMiddleware>();
        app.UseEndpoints(endpoints => { endpoints.MapControllers(); });
    }
}
```

5. Crearemos otro archivo llamado **ConfigureServiceExtension.cs** y hacemos una estructura como esta:

```

public static class ConfigureServicesExtension
{
    public static void InitConfigurationAPI(this IServiceCollection services, IConfiguration configuration)
    {
        services.AddApplicationLayer();
        services.AddPersistenceLayer(configuration);
        services.AddUnitOfWorkLayer();
        services.AddCommonLayer();
        services.AddTransient<ErrorHandlerMiddleware>();
        services.AddControllers().AddNewtonsoftJson(options =>
        {
            options.UseCamelCasing(true);
            options.SerializerSettings.Culture = System.Globalization.CultureInfo.CurrentCulture;
            options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore;
            options.SerializerSettings.NullValueHandling = Newtonsoft.Json.NullValueHandling.Ignore;
            options.SerializerSettings.DateTimeZoneHandling = Newtonsoft.Json.DateTimeZoneHandling.Local;
            options.SerializerSettings.FloatFormatHandling = Newtonsoft.Json.FloatFormatHandling.DefaultValue;
            options.SerializerSettings.FloatParseHandling = Newtonsoft.Json.FloatParseHandling.Decimal;
        });
    }
}

```

6. Agregamos la referencia de todos los proyectos de esta solución: **CA.Domain**, **CA.Infrastructure.Persistence**, **CA.Infrastructure.UnitOfWork**, **CA.Infrastructure.Common** y **CA.Application**.
7. En el archivo **CA.Api\StartUp.cs** hagamos el siguiente ajuste, para leer todas las inversiones de control traídos desde **CA.Api\ServiceCollection**:
8. Ahora, pasemos a la carpeta **CA.Api\Controllers**, vamos a modificar el archivo **ArticleController.cs** para que tome las implementaciones de MediatR para cada operación CRUD correspondiente. Debe ser así:

```

using System.Threading.Tasks;
using System.Collections.Generic;

using MediatR;
using Microsoft.AspNetCore.Mvc;

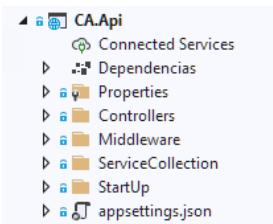
using CA.Domain.DTO;
using CA.Domain.Wrappers;
using CA.Application.Queries;

namespace CA.Api.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ArticleController : ControllerBase
    {
        private readonly IMediator _mediator;
        public ArticleController(IMediator mediator) => _mediator = mediator;
        [HttpGet]
        public async Task<IEnumerable<ArticleDTO>> GetArticles() => await _mediator.Send(new GetAllArticleQuery());
        [HttpGet("{id}")]
        public async Task<ArticleDTO> GetArticle(int id) => await _mediator.Send(new GetArticleQuery(id));
        [HttpPost]
        public async Task<ApiResponse<CreateArticleDTO>> Post(CreateArticleDTO command) => await _mediator.Send(command);
        [HttpPut]
        public async Task<ApiResponse<UpdateArticleDTO>> Put(UpdateArticleDTO command) => await _mediator.Send(command);
        [HttpDelete]
        public async Task<ApiResponse<DeleteArticleDTO>> Delete(DeleteArticleDTO command) => await _mediator.Send(command);
    }
}

```

Aquí notemos que ya estamos usando el objeto **IMediator**, injectado en el constructor de la **ApiController** y podemos enviar ahora sí, los objetos del tipo consulta y comando. Nótese que los verbos **POST**, **PUT** y **DELETE**, se devuelve un objeto del tipo **ApiResponse<T>**. Repetir esta estructura para los demás controladores de la Web Api.

9. Guardemos cambios y compilemos. Ahora, **CA.Api** debe tener una estructura así:



Con esto, ya tenemos ahora sí, nuestra solución mas limpia y estructurada y con código fuente más limpio y facil de hacer correcciones.

Si todo lo hicimos bien hasta aquí, nuestro proyecto debe funcionar mas o menos así:

```

[{"id": 0, "name": "Marca gen\u00e9rica", "supplierId": 0, "isSystemRow": true, "accountIdCreationDate": 0, "creationDate": "2022-02-16T23:18:16.473-06:00", "isDeleted": false}, {"id": 1, "name": "Gansito", "supplierId": 2, "isSystemRow": true, "accountIdCreationDate": 1}
]
  
```

Notemos que funciona correctamente y que ahora sí, tenemos un control de excepciones de manera limpia y ordenada. Nuestro proyecto está funcionando hasta ahora, de manera satisfactoria.

```

{
  "succeeded": false,
  "message": "One or more validation errors occurred",
  "errors": [
    {
      "shortName": [
        "Formato del nombre corto del art\u00edculo incorrecto."
      ]
    }
  ]
}
  
```

Si ejecutamos ahora la consulta de países (<http://localhost:44356/api/countrydetail>)... tenemos este detalle:

The screenshot shows a Postman interface with the following details:

- Request Method: GET
- URL: https://localhost:44356/api/countrydetail
- Params tab is selected.
- Query Params table:

KEY	VALUE	DESCRIPTION
Key	Value	Description
- Send button is visible.
- Response section shows a placeholder icon and the message "Could not get response".
- An error message box is displayed: "Error: Maximum response size reached | View in Console".
- A link below the error message says "Learn more about troubleshooting API requests".

Lo cual, es un problema por que el JSON generado es muy grande y se tiene que controlar con una paginación de registros, cosa que veremos más adelante en este documento.

9.11. Conclusión.

Finalmente, nuestro proyecto va tomando forma con la implementación de los patrones de diseño CQRS y Mediator y el uso del Middleware de .NET Core para controlar las excepciones de validación y las demás excepciones que ocurran dentro de nuestra aplicación Web API.