Clean Architecture in .NET Core: Step by Step

Olimpo Bonilla Ramírez

TABLA DE CONTENIDO

INTRODUCCIÓN.

Este documento electrónico tiene como objetivo, explicar paso a paso como desarrollar una aplicación en .NET Core con el módelo 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 empresaspara 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 tambien, 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, administramos 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 Corepara 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 procecimiento 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.*

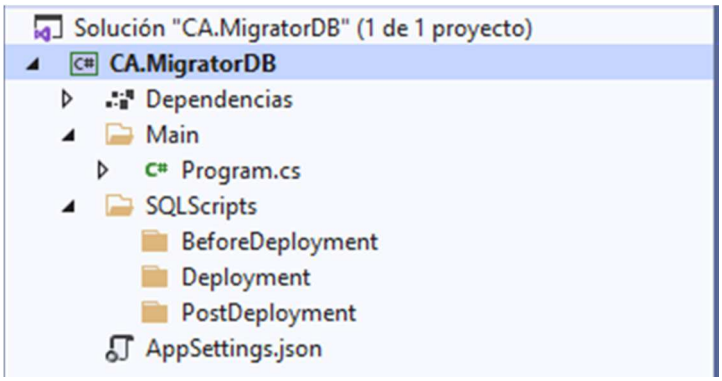
1.4. Uso de DbUp para Fluent Migration DataBase.

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 automaticamente 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($"CollectionConnectionStrings").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($"CA.MigratorDB.SQLScripts.BeforeDeployment."),
                new SqlScriptOptions { ScriptType = ScriptType.RunAlways, RunGroupOrder = 0 })
              // Main Deployment scripts: se ejectan solo una vez y corren en el segundo grupo.
              .WithScriptsEmbeddedInAssembly(Assembly.GetExecutingAssembly(), x =>
                x.StartsWith($"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 lineas:

```
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 órden de ejecución de los scripts en el proceso de migración. Se empieza desde 0.
Las siguientes líneas

```
// Main Deployment scripts: se ejectan 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 órden 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.

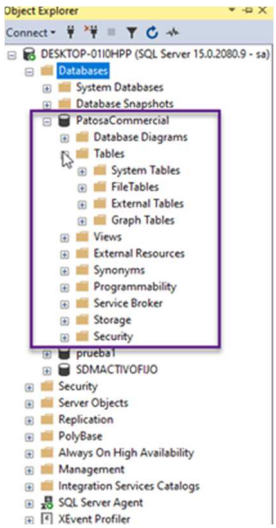
```
if (upgrader.IsUpgradeRequired())
{
  var result = upgrader.PerformUpgrade();

  /* 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": {
    "DefaultConnectionBD": "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 compilemos. 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 eliminen 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 teniendolo así, hace dificil su rastreabilidad, a la hora de buscar referencia historica 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 tambien, hay fanaticos 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 fanaticos 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:

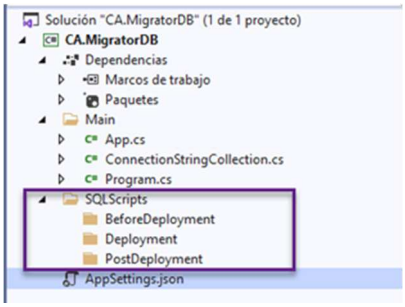
   Oye, estoy terminando de agregar el número de cuenta a nuestros clientes, pero no sé qué deberíamos poner allí para los existentes. ¿Debería ser algún tipo de N/A que nuestros usuarios administradores deben completar más tarde o desea que los complete por adelantado con el archivo que me proporcionará?

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 ejecútelos. 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 creemos 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.

   - 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.

   - 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.

   - Creamos la tabla de sucursales.

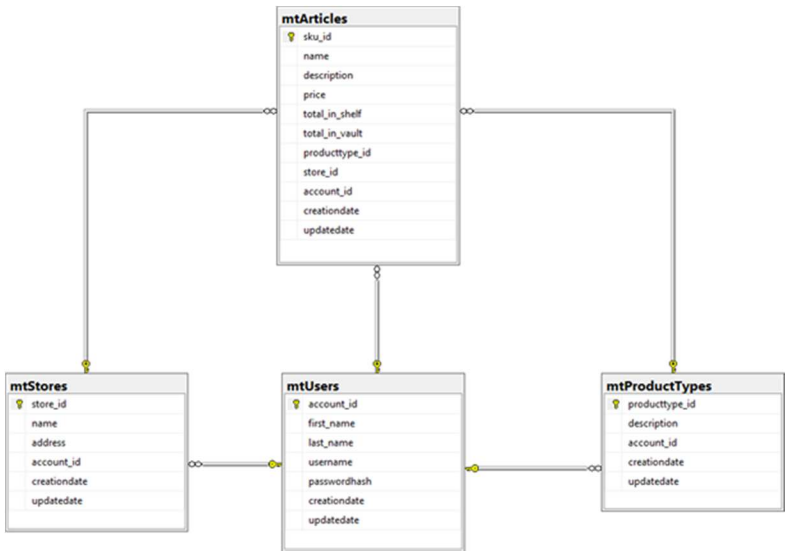
Campo:	Descripción:
store_id	Identificador de la sucursal.
name	Nombre de la sucursal.
address	Domicilio o dirección de la sucursal.
account_id	Identificador de la cuenta de usuario que hizo el alta del registro.
creationdate	Fecha de alta.
updatedate	Fecha de actualización.

   - 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.
-------------------------------------
IF OBJECT_ID('dbo.mtUsers') IS NOT NULL
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;

  IF OBJECT_ID('dbo.mtUsers') IS NOT NULL
    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

IF OBJECT_ID('dbo.mtUsers') IS NOT NULL
  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
```

Tabla de artículos, **mtArticles.**

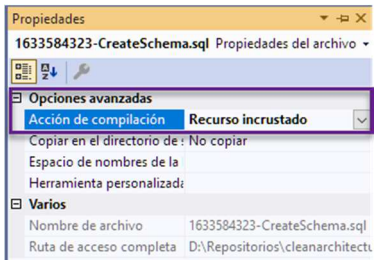
```
-- 4. Catálogo de artículos.
-----------------------------
IF OBJECT_ID('dbo.mtArticles') IS NOT NULL
BEGIN
  DROP TABLE dbo.mtArticles;

  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),
  CONSTRAINT       [fk_Idarticle1]          FOREIGN KEY(account_id) REFERENCES mtUsers(account_id),
  CONSTRAINT       [fk_Idarticle2]          FOREIGN KEY(store_id) REFERENCES mtStores(store_id),
  CONSTRAINT       [fk_Idarticle3]          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.**

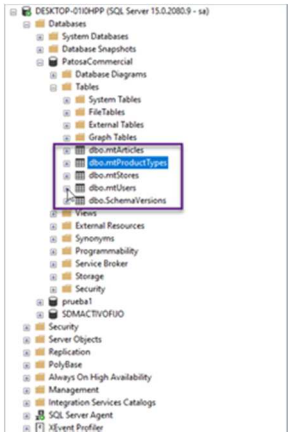
4. Por último, la carpeta **PostDeployment** es para probar que todos los componentes u objetos de Base de Datos creados anteriormente funcionen. Esta carpeta, a veces no la usamos, pero si debemos tener en cuenta que probando primero, asegura el éxito de la migración fluida en Base de Datos.
5. Finalmente, 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 (Embebedd 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 automaticamente 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 que parte del deployment en Base de Datos ocurre esto? Lo dejo 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-CheckSP.sql	2021-10-07 02:09:29.883
6	6	CA.MigratorDB.SQLScripts.PostDeployment.1633590409-CheckSP.sql	2021-10-07 02:10:17.470

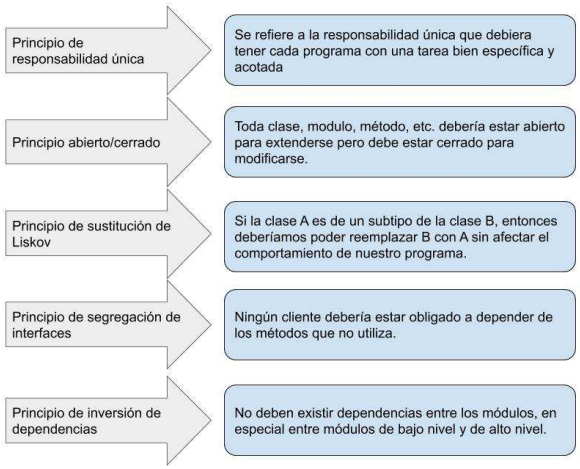
Con esto, terminamos nuestro proyecto de migración de Base de Datos, esencial para nuestro tutorial, en los capítulos siguientes.

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 esta 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 Subsistution 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 especificas 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 Princple (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 cambiable; 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.

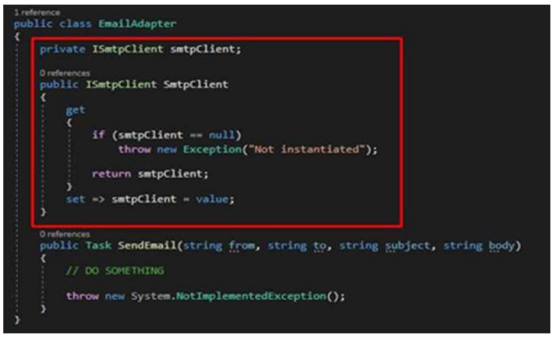
Se trata de extraer responsabilidades a un componente para delegarlas a otros componentes, de tal manera que cada componente solo tiene una responsabilidad (Principio de Responsabilidad Única de SOLID). Estas responsabilidades pueden cambiarse en tiempo de ejecución sin ver alterado el resto de comportamientos.

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 mantenible 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 mas 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.



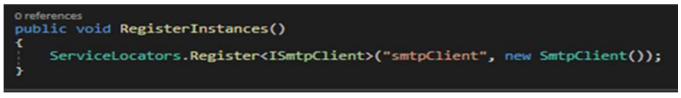
- **Por constructor (Constructor).** Cuando el constructor de una clase recibe una abstracción de otra clase como parámetro. Es el más comunmente usado.

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

- **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 especifica.

  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.

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



  Finalmente, podemos usarlo así:

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

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

    0 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:

```
0 referencias | Olimpo Bonilla Ramírez, Hace 4 días | 1 autor, 1 cambio
static async Task Main(string[] args)
{
    var services = ConfigureServices();
    var serviceProvider = services.BuildServiceProvider();
    await serviceProvider.GetService<App>().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 IServiceCollection ConfigureServices()
{
    IServiceCollection 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));

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

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

    return services;
}
```

IServiceProvider

IServiceCollection

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:

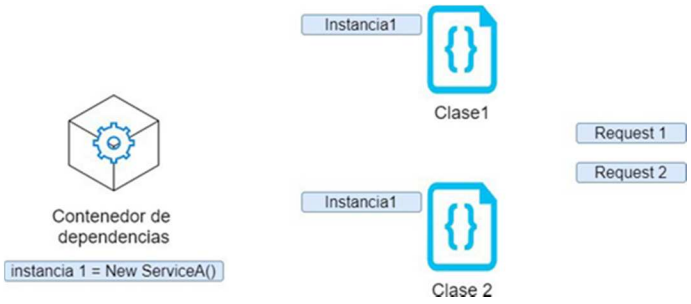
- El primer argumento es nuestra **abstracción,** o sea, la interfaz que vamos a utilizar a la hora de inyectar en los diferentes constructores.

- 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.

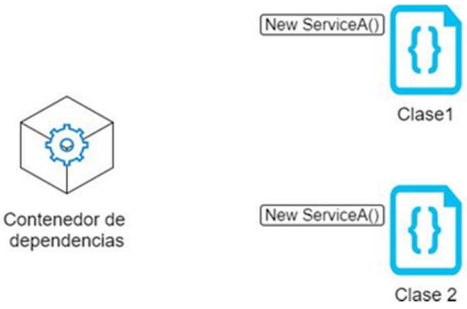
- **Scoped.** Los servicios se crearan **una vez** por solicitud del cliente.



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.

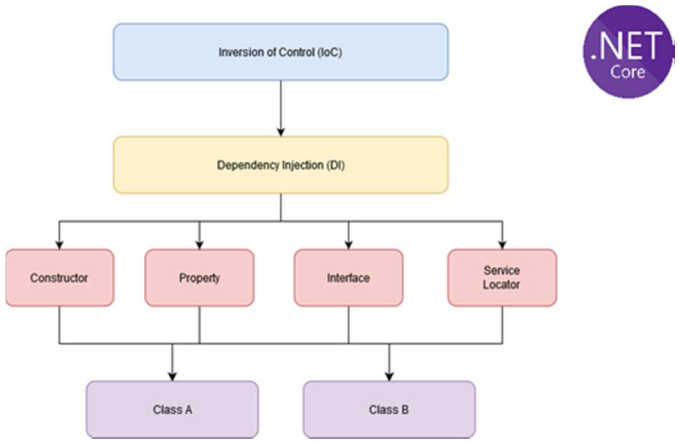
```
services.AddTransient<ILog, MyLog>();
/* O bien... */
services.AddTransient(typeof(ILog), typeof(MyLog));
```

- **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.

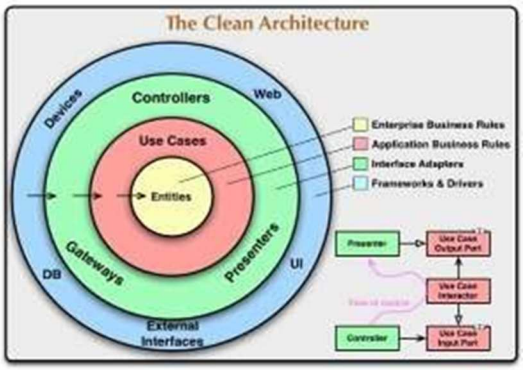
Con esto tenemos entonces comprendido el concepto de Inyección de Dependencias e Inversión de Control, para ya entrar de lleno al concepto de *Clean Architecture.*

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.

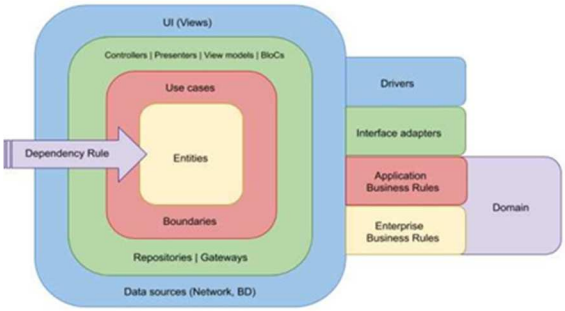
Regla de Dependencia. Para entender el diagrama tenemos que utilizar la regla de la dependencia:

"Las dependencias solo pueden apuntar hacia adentro. Nada en un circulo interno puede conocer algo de un circulo externo. El nombre de algo declarado en un circulo externo, no puede ser mencionado en un circulo 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 circulo interior nunca debe conocer nada sobre un circulo 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 ventas 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.

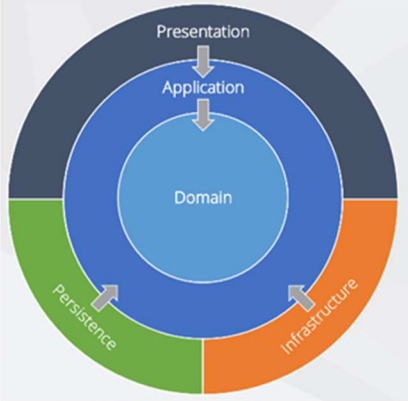
¿Cuándo usar Clean Architecture? En general, si se decide usar este patrón de diseño, debe considerarse el uso. Tomemos en cuenta estas consideraciones.

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 módelo 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 líneamientos 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 módelo. 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 infrastructura, 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 infrastructura, 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 infrastructura y deben implementar interfaces definidas en la capa de Core, por lo que la infrastructura deberá tener una referencia a la capa de Domain. Los tipos de infrastructura son:

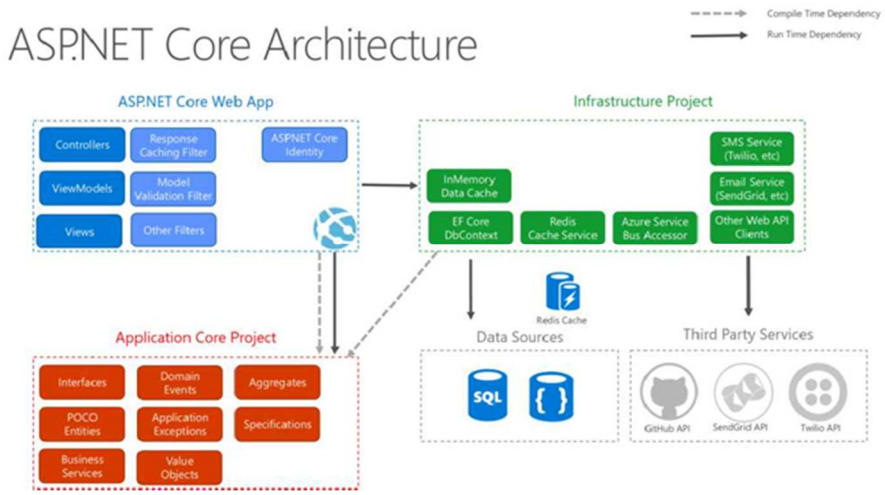
- Tipos de EntityFramework Core (DbContext, Migration).
- Tipos de impkementación de acceso a datos (Repository).
- Servicios especificos de la infrastructura (por ejemplo FileLoggerService).

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.

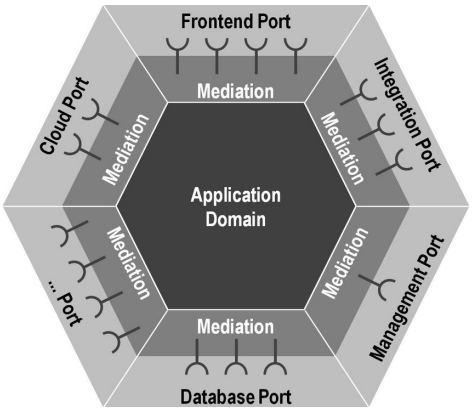
La clase **Startup** es responsable de configurar la aplicación y de conectar los tipos de implementación a las interfaces, lo que permite que la inserción de dependencias funcione correctamente en tiempo de ejecución.

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.

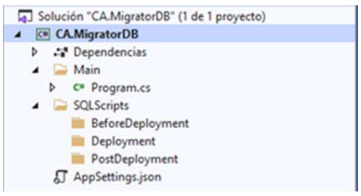
Una vez ya teniendo en claro estos conceptos, vamos entonces a implementar nuestro proyecto en Visual Studio. Necesitamos reorganizar nuestro proyecto inicial, puesto que hasta ahora, solo tenemos la aplicación de consola de Migrator DataBases (**CA.MigratorDB**). Entonces, pensando bien las cosas, decidimos usar el módelo *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.

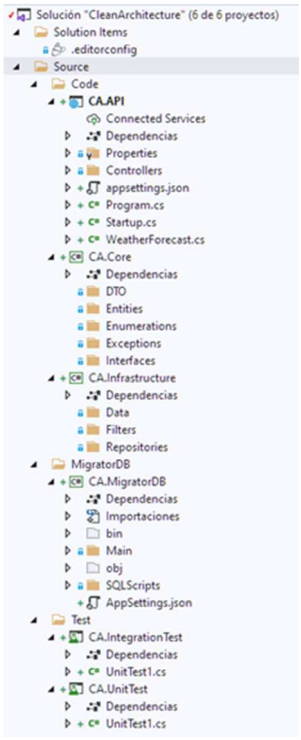
Entonces armaremos nuestro proyecto anterior como sigue:

1. Nuestro proyecto venia anteriormente así:

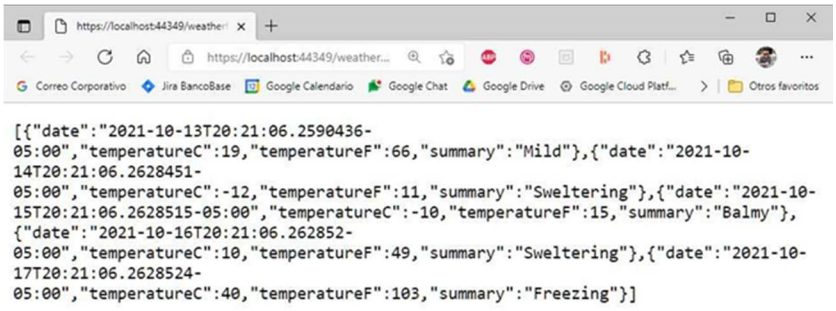


2. Quitemos el proyecto llamado **CA.MigratorDB** por un momento y abramos fisicamente la carpeta de la solución **CleanArchitecture.sln**.
3. Creamos ahí tres nuevas carpetas llamadas **Test, MigratorDB y Code.** Pasemos el proyecto de migración de Base de Datos llamado **CA.MigratorDB** a la carpeta 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 fisica 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 fisica 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 fisicamente 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 fisicamente 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 demas 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:



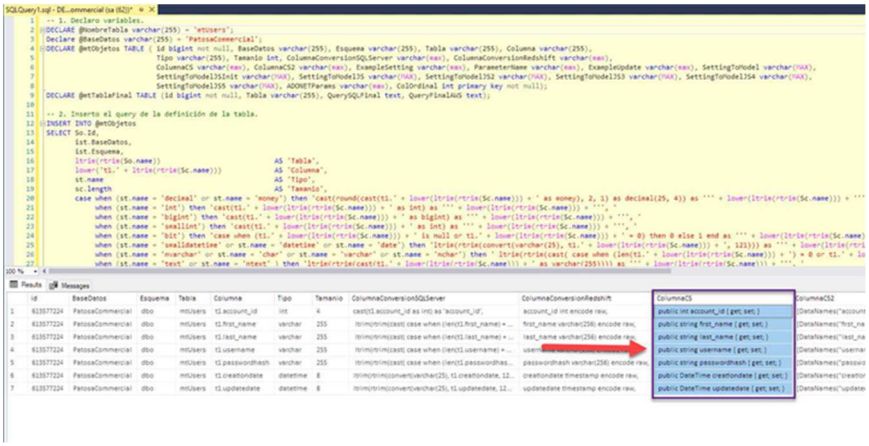
Lo cual indica que este proyecto está funcionando bien.

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.**

1. 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 mtStores
    {

        public int store_id { get; set; }
        1 referencia | 0 cambios | 0 autores, 0 cambios
        public string name { get; set; }
        1 referencia | 0 cambios | 0 autores, 0 cambios
        public string address { get; set; }
        1 referencia | 0 cambios | 0 autores, 0 cambios
        public int account_id { get; set; }
        1 referencia | 0 cambios | 0 autores, 0 cambios
        public DateTime creationdate { get; set; }
        1 referencia | 0 cambios | 0 autores, 0 cambios
        public DateTime? updatedate { get; set; }
    }
}
```

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

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

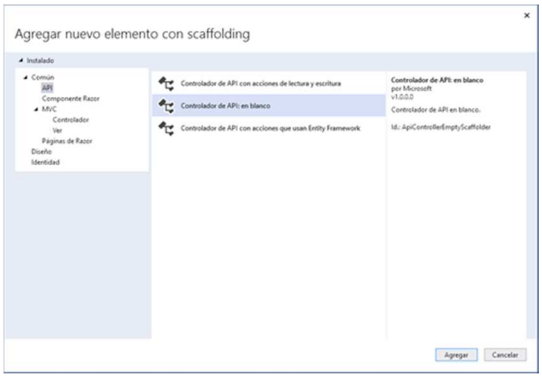
1. En la carpeta **CA.Infrastructure\Repositories** crear una clase llamada **StoreRepository.cs.** El contenido de este archivo es el siguiente.

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

namespace CA.Infrastructure.Repositories
{
    1 referencia | 0 cambios | 0 autores, 0 cambios
    public class StoreRepository
    {
        1 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;
        }
    }
}
```

2. Guardemos los cambios y compilamos. Para efectos de enseñanza, haremos datos dummie. 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:

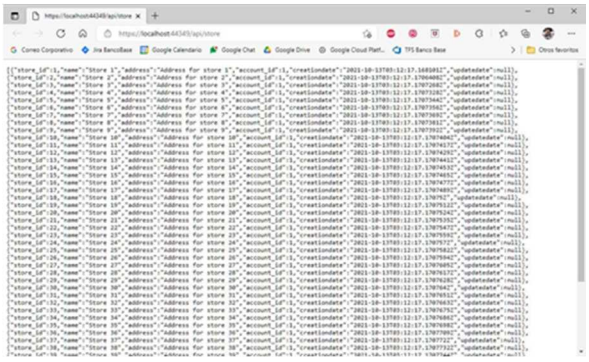
1. En la carpeta **CA.Api\Controllers** crear un controlador llamado **StoreController.cs.** Lo podemos crearlo sobre la carpeta *Controllers* y pulsando el menú flotante, elegimos la ruta **Agregar\Controller** para que nos aparezca la siguiente ventana:



   Elegimos *Controlador de API en blanco,* pulsamos *Agregar* y escribimos el nombre del controllador como *StoreController.* Pulsamos *Aceptar* y se generará el controlador en esa carpeta.
2. Eliminemos el archivo **WeatherForecastController.cs** de esa carpeta antes mencionada.
3. Eliminemos el archivo **WeatherForecast.cs** de la carpeta raíz del proyecto de **CA.Api.**
4. Nuestro archivo llamado StoreController.cs debe tener algo como esto:

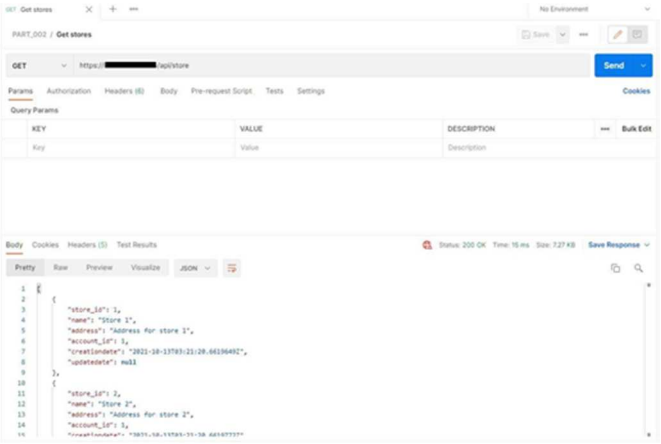
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using CA.Infrastructure.Repositories;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace CA.Api.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    0 referencias | 0 cambios | 0 autores, 0 cambios
    public class StoreController : ControllerBase
    {
        [HttpGet]
        0 referencias | 0 cambios | 0 autores, 0 cambios
        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.
5. 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**.
6. 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 creemos 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 cua, 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:
`IArticleService`	`ArticleService`
`IArticleRepository`	`ArticleRepository`
`INotifier`	`EmailNotifier`

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 scalfolding 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 conexto 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> GetArticles(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 seria `UseMySQL` o para PostgreSQL seria `UsePosteSQL`. Con esto, ya tenemos listo, a nivel de ambito 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.