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).

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).

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.*

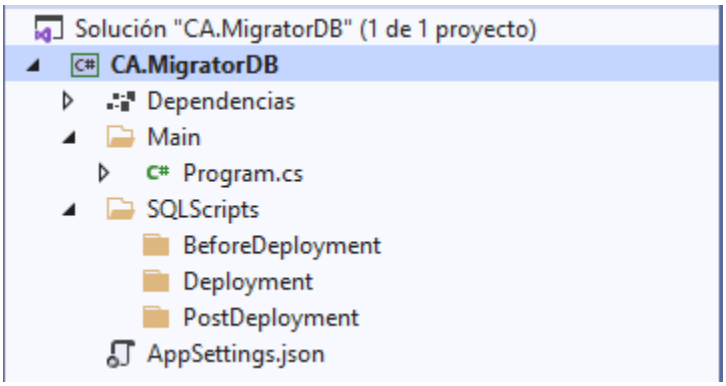
1.4. Uso de DbUp para Fluent Migration DataBase.

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 *EF 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*, 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 LoadConfiguration()
    {
      var builder = new ConfigurationBuilder().SetBasePath(Directory.GetCurrentDirectory())
                                              .AddJsonFile("AppSettings.json", optional: true,
                                                           reloadOnChange: true);
      return builder.Build();
    }

    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;
    }
  }
}
```

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();
}
```

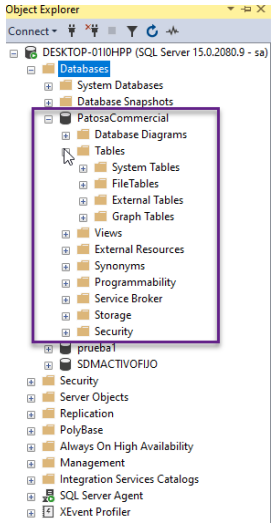
Este bloque finalmente hace la conclusión del proceso de migración de Base de Datos. La función **PerformUpgrade** ejecuta todo el lote de scripts almacenados en el objeto **upgrader** y genera un resultado. Si es **Successful**, quiere decir que se aplicaron correctamente los cambios en la Base de Datos, 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:

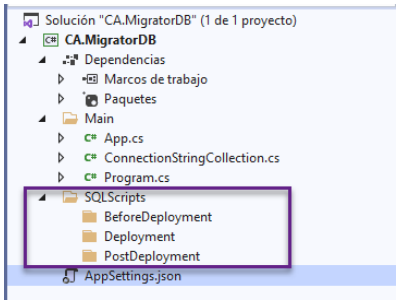
   Oye, estoy terminando de agregar el número de cuenta a nuestros clientes, pero no sé qué deberíamos poner allí para los existentes. ¿Debería ser algún tipo de N/A que nuestros usuarios administradores deben completar más tarde o desea que los complete por adelantado con el archivo que me proporcionará?

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 capitulo, 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.

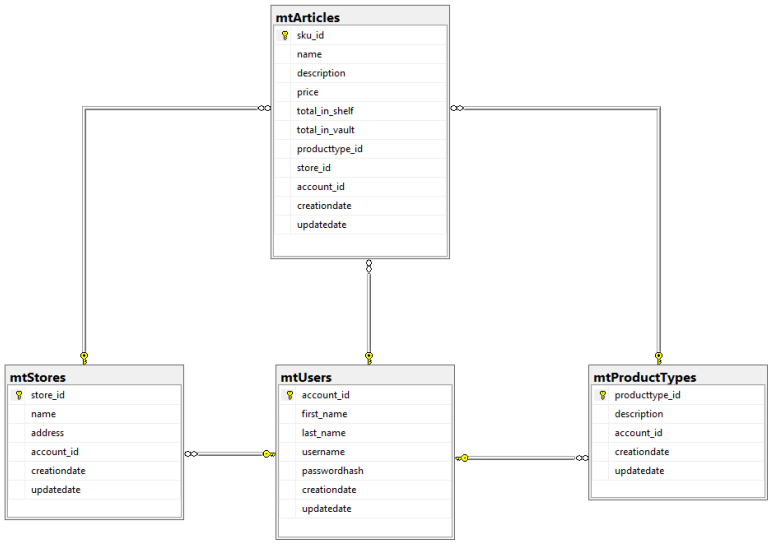
- Creamos la tabla de sucursales.

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
  PRINT '<<< Ocurrió un error al crear el objeto ''dbo.mtStores''. >>>';
```

Tabla de artículos, **mtArticles.**

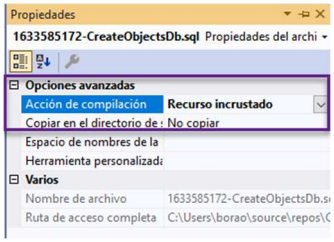
```
-- 4. Catálogo de artículos.
-----------------------------
IF OBJECT_ID('dbo.mtArticles') IS NOT NULL
BEGIN
  DROP TABLE dbo.mtArticles;

  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),
  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]-LoadTables.sql**

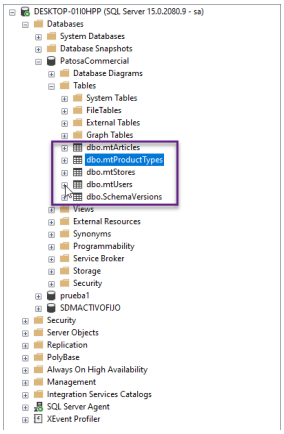
4. Por último, la carpeta **PostDeployment** es para probar que todos los componentes u objetos de Base de Datos creados anteriormente funcionen. Esta carpeta, a veces no la usamos, pero si debemos tener en cuenta que probando primero, asegura el éxito de la migración fluida en Base de Datos.
5. Finalmente, 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.