Créditos Directos S.A.
Casa Central: Ituzaingó 1315 Montevideo Uruguay
Tel: (2) 18918

TÍTULO

Arquitectura y Guía de proyectos .NET CORE 2.2 y GRAPHQL

Versión 1.1

Control de Cambios:

Versión	Fecha	Autor	Cambios
1.0	02/08/19	Pedro Cadevilla	Elaborar documento.
2.0	01/10/19	Pedro Cadevilla	Se agregó el uso y creación de proyectos

Creación de solución y proyectos

Directrices principales para la generación de servicios .NET

El nombrado de la solución debe ser alusivo al desarrollo interno que se va a hacer del proyecto usando PascalCase (PascalCase será el estándar de nombrado para todos los archivos y variables dentro del proyecto). **Ejemplo:**

Nombre de solución: WorkflowGraphQL

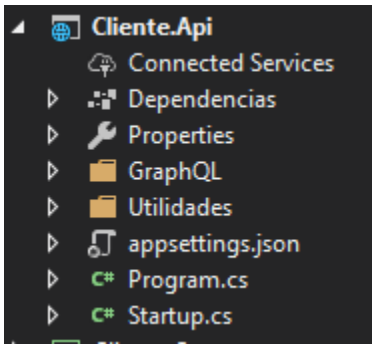
Dentro de la solución se deben generar 4 proyectos básicos alusivos al contenido definidos en su extensión por la funcionalidad que van a presentar y estarán divididos en capas como lo son API, Core, Persistence y CommonUtils los cuales se definirán a continuación.

Workflow.Api:

Es la capa donde se van a desarrollar todos los llamados servicios para el funcionamiento general del proyecto. Su arquitectura interna general es la siguiente:

1. El tipo de proyecto a crear es una **"Aplicación web ASP.NET Core"** usando NET Core 2.2**.**

2. Dentro se debe crear una carpeta llamada **GraphQL** donde se van a ubicar las clases y componentes del servicio y una carpeta **Utilidades** donde se tendrán los métodos o funciones comunes para la lógica de los servicios.



3. Dicho proyecto tendrá 2 clases llamadas **Startup.cs** y **Program.cs**, su funcionalidad es la siguiente:

   - **Startup.cs:** Encargada de la configuración de inicio de servicios.
   - **Program.cs:** Encargada de la creación del HostWeb.

4. La carpeta **GraphQL** estará conformada por las siguientes subcarpetas:

InputTypes:

- El estándar de nombrado de los archivos de dicha carpeta se hace usando el nombre de la tabla usando de prefijo la palabra clave **InputType**. **Ejemplo:** CodigoAreaInputType.cs

- Es donde se albergara toda la configuración del tipo de dato SOLICITADO es decir el INPUT de entrada por modelo o tabla.

- Se definirá una propiedad por campo llamada **Description** que recibe un parámetro de tipo texto donde se define el campo de la tabla.

- Se definirá también el tipo de dato a recibir con la propiedad **Type** pudiendo ser el mismo no nulo asignándole la definición **NoNullType** o **Nullable** si no se le asigna la misma.

Ejemplo:

```
public class CodigoAreaInputType : InputObjectType<CodigoArea>{
       protected override void Configure(IInputObjectTypeDescriptor<CodigoArea>
descriptor){
            descriptor.BindFields(BindingBehavior.Explicit);
            descriptor.Name("CodigoAreaInput");

            descriptor.Field(x => x.Nombre).Description("Nombre de Codigo");
            descriptor.Field(x =>
x.Codigo).Type<NonNullType<StringType>>().Description("Codigo numerico de area");
        }
}
```

Mutations:

- Es donde se van a crear todos los métodos que consistan en interactuar con la BD de alguna manera (Crear, Modificar, Eliminar) por modelo o tabla.

- Las clases serán creadas usando **Partial Classs** donde el archivo principal llamado **Mutations.cs** contiene la información común de todos y se definirán archivos únicos por modelo o tabla.

- El estándar de nombrado de los archivos de dicha carpeta se hace usando el nombre de la tabla agregándole de prefijo la palabra clave **Mutation**. **Ejemplo:** PersonaNaturalMutation.cs

```
public partial class Mutation{
        public async Task<PersonaNatural> CrearPersonaNaturalAsync(PersonaNatural
persona)
        {
            _unitOfWork.PersonaNatural.Add(persona);
            await _unitOfWork.Complete();
            return persona;
        }

        public async Task<PersonaNatural> ModificarPersonaNaturalAsync(long id)
        {
            PersonaNatural personaActual = await
_unitOfWork.PersonaNatural.ObtenerPorClienteIdAsync(id);
            persona.Id = id;
            if (personaActual == null)
            {
                throw new ArgumentException("Elemento no encontrado en la base de
datos");
            }

            var personaNueva = JsonConvert.SerializeObject(persona,
Newtonsoft.Json.Formatting.None,
new JsonSerializerSettings
{ NullValueHandling = NullValueHandling.Ignore });
            JsonConvert.PopulateObject(personaNueva, personaActual);
            await _unitOfWork.Complete();
            return personaActual;
        }

        public async Task<PersonaNatural> EliminarPersonaNaturalAsync(long id)
        {
            PersonaNatural personaActual = await
_unitOfWork.PersonaNatural.ObtenerPorClienteIdAsync(id);
```

```
            if (personaActual == null)
            {
                throw new ArgumentException("Elemento no encontrado en la base de
datos");
            }

            _unitOfWork.PersonaNatural.Delete(personaActual);
            await _unitOfWork.Complete();
            return personaActual;
        }
}
```

Queries:

- Es donde se ubicaran las consultas necesarias para los procesos.

- Las clases serán creadas usando **Partial Classs** donde el archivo principal llamado **Query.cs** contiene la información común de todos y se definirán archivos únicos por modelo o tabla.

- El estándar de nombrado de los archivos de dicha carpeta se hace usando el nombre de la tabla agregándole de prefijo la palabra clave **Query**. **Ejemplo:** PersonaNaturalQuery.cs

```
public partial class Query{
        public async Task<PersonaNatural> ObtenerCodigoCliente(string cedula)
        {
            if (cedula != null)
            {
                IdentificacionPersonaNatural identificacion = await
_unitOfWork.IdentificacionPersonaNatural.ObtenerPorNumeroAsync(cedula);
                return await
_unitOfWork.PersonaNaturalCache.ObtenerPorClienteIdAsync(identificacion.ClienteId);
            }
            return null;
        }
}
```

Querytypes:

- El estándar de nombrado de los archivos de dicha carpeta se hace usando el nombre de la tabla más la palabra clave **Type** luego del nombre de la misma. **Ejemplo:** NacionalidadType.cs

- Es donde encontraremos la especificación de datos y posibles relaciones que puedan devolver en las consultas por modelo o tabla.

- Se definirá una propiedad por campo llamada **Description** que recibe un parámetro de tipo texto donde se define el campo de la tabla.

- Se pueden definir campos **Resolvers** los cuales son los encargados de devolver subconsultas dentro del query estando los mismos programados en la capa de negocios (**Persistence**).

```
public class NacionalidadType : ObjectType<Nacionalidad>{
        protected override void Configure(IObjectTypeDescriptor<Nacionalidad>
descriptor)
        {
            descriptor.BindFields(BindingBehavior.Explicit);
            descriptor.Description("Nacionalidades de la BD");

            descriptor.Field(x => x.Id).Description("ID de Nacionalidad");
            descriptor.Field(x => x.Nombre).Description("Nombre de la
Nacionalidad");
            descriptor.Field(x => x.NombreAlternativo).Description("Nombre
Alternativo de la Nacionalidad");
            descriptor.Field<NacionalidadResolver>(x =>
x.ObtenerPersonasAsync(default)).Description("Personas que Pertenecen a la
Nacionalidad");
        }
}
```

Resolvers:

- El estándar de nombrado de los archivos de dicha carpeta se hace usando el nombre de la tabla más la palabra clave **Resolver** luego del nombre de la misma. **Ejemplo:** NacionalidadResolver.cs

- Es donde se ejecuta el llamado a consultas compuestas previamente realizadas en el **Persistence** y que serán ejecutadas desde su llamada en el **API** a través del Query respectivo**.**

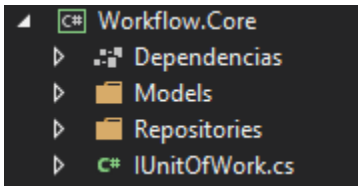
```
public class NacionalidadResolver{
        protected readonly IUnitOfWork _unitOfWork;

        public NacionalidadResolver(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }

        public async Task<IEnumerable<PersonaNatural>>
ObtenerPersonasAsync([Parent]Nacionalidad nacionalidad)
        {
            return await
_unitOfWork.Nacionalidad.ObtenerPersonasAsync(nacionalidad.Id);
        }
}
```

5. La carpeta **Utilidades** estará conformada por clases nombradas en formato PascalCase y siendo sus nombres alusivos a la funcionalidad común ejecutada dentro de la misma. **Ejemplo:** Comparador.cs

Workflow.Core:

Es la capa donde se encuentra el modelo de datos de las tablas y las interfaces de las mismas generadas dado el comportamiento presentado en la capa PERSISTENCE.

1. El tipo de proyecto a crear es **"Biblioteca de clases (.NET Core)".**



2. Dentro del proyecto debe haber un archivo de interfaz llamado **IUnitOfWork.cs** donde se alojaran las referencias de las interfaces necesarias para hacerlas accesibles desde **API**.

```
public interface IUnitOfWork : IDisposable{
        INacionalidadRepository Nacionalidad { get; }
        ICodigoAreaRepository CodigoArea { get; }
        ICodigoPaisRepository CodigoPais { get; }
        IEstadoCivilRepository EstadoCivil { get; }
        IGeneroRepository Genero { get; }
        IHorarioContactoRepository HorarioContacto { get; }
        IPersonaNaturalRepository PersonaNatural { get; }
        Task<int> Complete();
}
```

3. Dentro se deben crear 2 carpetas:

- **Models:** Es donde se generara el modelo de las tablas de existir la BD usando **SCAFFOLDING** en caso contrario se debe generar directamente desde código para migrar luego a la BD. El estándar de nombrado de los archivos de dicha carpeta se hace usando el nombre de la tabla como nombre de la clase. **Ejemplo:** Nacionalidad.cs

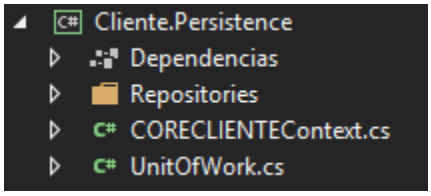
- **Repositories:** Es donde se alojaran las interfaces de repositorio que serán las encargadas de manejar las definiciones de las funciones por tabla desarrolladas en la capa de persistencia. El estándar de nombrado de los archivos de dicha carpeta se hace usando el nombre de la tabla entre la letra "i" mayuscula más la palabra clave **Repository** luego del nombre de la misma. **Ejemplo:** INacionalidadResolver.cs

```
public interface IIdentificacionPersonaNaturalCacheRepository :
IRepository<IdentificacionPersonaNaturalCache>{
        Task<IdentificacionPersonaNaturalCache>
ObtenerPorClienteIdAsync(long? id);
        Task<IdentificacionPersonaNaturalCache> ObtenerPorNumeroAsync(string
cedula);
}
```

Cliente.Persistence:

Es la capa donde se desarrollan las consultas y funciones de las tablas definidas previamente en la capa CORE para ser implementadas por la capa **API** a través del uso del **UnitOfWork**.

1. El tipo de proyecto a crear es "Biblioteca de clases (.NET Core)".



2. Dentro del proyecto debe haber un archivo de clase llamado **UnitOfWork.cs** donde se alojaran las definiciones necesarias para cada uno de los repositorios (manejadores de modelo o clase) para ser expuestos por la interfaz.

```
public class UnitOfWork : IUnitOfWork{

        private readonly CORECLIENTEContext _Context;
        public INacionalidadRepository Nacionalidad { get; private set; }
        public ICodigoAreaRepository CodigoArea { get; private set; }

        public UnitOfWork(CORECLIENTEContext Context){
            _Context = Context;
            Nacionalidad = new NacionalidadRepository(_Context);
            CodigoArea = new CodigoAreaRepository(_Context);
        }

        public async Task<int> Complete(){
            return await _Context.SaveChangesAsync();
        }

        public void Dispose(){
            _Context.Dispose();
        }
}
```

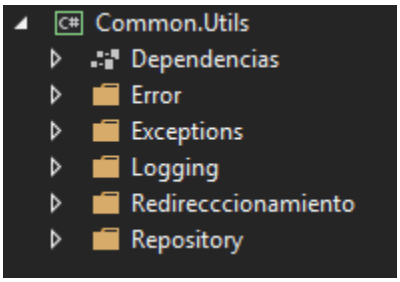
1. También se debe colocar aquí el archivo de Contexto ya sea generado por el **SCAFFOLDING** o generado propiamente el cual es quien tiene la definición de las relaciones, restricciones y demás propiedades de las tablas. El estándar de nombrado de los archivos de contexto se hace usando el nombre de la base de datos como nombre de la clase más la palabra clave **Context** luego del nombre de la misma. **Ejemplo:** CORECLIENTEContext.cs

2. Dentro de la carpeta **REPOSITORIES** se encontraran todos los repositorios de funciones o consultas por clase en donde se manejara toda la lógica de negocios. El estándar de nombrado de los archivos de contexto se hace usando el nombre de la base de datos como nombre de la clase más la palabra clave **Repository** luego del nombre de la misma. **Ejemplo:** NacionalidadRepository.cs

```
class NacionalidadRepository : Repository<Nacionalidad>, INacionalidadRepository{
        public NacionalidadRepository(DbContext context) : base(context)
        { }

        public CORECLIENTEContext CORECLIENTEContext
        {
            get { return _Context as CORECLIENTEContext; }
        }

        public async Task<IEnumerable<PersonaNatural>> ObtenerPersonasAsync(long
nacionalidadID)
        {
            if (nacionalidadID == 1)
            {
                throw new BussinessException("1");
            }
            IEnumerable<PersonaNatural> personas = await
CORECLIENTEContext.PersonaNatural
            .Where(x => x.NacionalidadId == nacionalidadID)
            .ToListAsync();
            return personas;
        }
}
```

CommonUtils:

Es el proyecto encargado de manejar los encapsulados de respuesta, errores, excepciones y logging de toda la solución así como manejar decodificaciones, encriptaciones u operaciones básicas comunes. (Este proyecto esta previamente definido y publicado en repositorio en consenso general).



El proyecto está compuesto por 5 carpetas principales las cuales indican claramente su contenido y su contenido está definido de la siguiente forma:

Error:

- Contiene 2 archivos un filtro de errores **ErrorFilter.cs** y un tipo de mensaje de error **ErrorMessageType.cs** que no es más sino el estándar a ser devuelto por la solución en caso de fallo dichos archivos contienen los códigos de error respectivos con su manejo de mensaje.

Exceptions:

- Contiene la clase de excepciones personales (negocio) con sus respectivos constructores.

- Para usar estas excepciones personales se deben cumplir 2 pasos el primero es ubicar la excepción en la capa de Persistencia dada la condición requerida usando los constructores ubicados en la clase **BussinessException.cs**.

```
public async Task<IEnumerable<PersonaNatural>> ObtenerPersonasAsync(long
nacionalidadID){
            if (nacionalidadID == 1)
            {
                throw new BussinessException("ID1");
            }
            else
            {
                throw new BussinessException("ID2", "Valor1", "Valor2");
            }
            IEnumerable<PersonaNatural> personas = await
CORECLIENTEContext.PersonaNatural
            .Where(x => x.NacionalidadId == nacionalidadID)
            .ToListAsync();
            return personas;
}
```

- Luego de esto se debe agregar el mensaje con su especificación de **ID** en el archivo **appsettings.json** dentro del bloque **BussinessCodes** de la siguiente manera (Se debe tomar en cuenta que dicho bloque de mensajes de excepciones de negocio debe ser general para todos los proyectos permitiendo así usar el mismo estándar de mensaje por tipo de erro).

```
  "BussinessCodes": {
      "ID1": "USUARIO SIN PERMISOS",
      "ID2": "USUARIO NO VALIDADO"
    },
```

Logging:

- Incluye la clase **DiagnosticObserves.cs** encargada de manejar los eventos ocurridos en la app y la clase **Headers.cs** que se encarga de controlar los datos recibidos para ser mostrados en log.

Redireccionamiento:

- Contiene los métodos para el llamado a otros proyectos tanto REST como GraphQL.

- Para usar el re direccionamiento hacia GraphQL se deben seguir las siguientes pautas. Se debe crear el **Query** o **Mutation** en forma de string tal como se muestra a continuación para el caso de la mutation se debe crear también el objeto nuevo a actualizar. Ademas de esto se debe dar una descripción del llamado que se hace y una dirección hacia donde apuntara el llamado.

```
public async Task<IEnumerable<CatNacionalidad>> ObtenerPersonasAsyncExterno(){
            string Query = @"
                query obtenernacionalidades(filtro:" + Variable + @" ){
                nombreNacionalidad,
                catNacionalidadId,
                nombreAlternativo,
                estadoRegistro
                obtenerPersonas{
                    nombre1
                }
                }";
            var Result = await Llamada.QueryAsync<CatNacionalidad>(Query,
"Descripcion", "DireccionID1");
            return Result;
}

public async Task<CatNacionalidad> CrearNacionalidadAsyncExterno(CatNacionalidad
Nacionalidad){
            string Query = @"
                mutation ($Nacionalidad: CatNacionalidad!){
                crearNacionalidad(nacionalidad: $Nacionalidad){
                    nombreNacionalidad,
                    nombreAlternativo,
                    estadoRegistro
                }
                }";
            var Result = await Llamada.MutationAsync<CatNacionalidad>(Query,
"Descripcion", "EndPoint", Objeto, "DireccionID2");
            return Result;
}
```

- Luego de esto se debe agregar la direccion con su especificación de **ID** en el archivo **appsettings.json** dentro del bloque **DirectionCodes** de la siguiente manera (dichos valores podrán ser cambiados a nivel de deploy para re direccionar a otra dirección y solo se debe reiniciar el servicio correspondiente para que el cambio se haga efectivo).

```
"DirectionCodes": {
    "DireccionID1": "https://localhost:5001/",
    " DireccionID2": "https://localhost:5091/",
  },
```

- Para usar el re direccionamiento hacia GraphQL se deben seguir las siguientes pautas. Se debe llamar al método de la clase Calls llamado **PostRestAsync** el cual recibe la direccionID como primer parámetro seguido del endpoint requerido para esa dirección base sumado de un JSON en formato de string y el token de aplicación.

```
public async Task<IEnumerable<PersonaNatural>> ObtenerPersonasAsync(int
nacionalidadID){
            var Result = Llamada.PostRestAsync("DireccioID", "EndPoint", "JSON en
formato String", "Token");
            if (Result != null)
            {
                throw new BussinessException("1");
            }
            else
            {
                IEnumerable<PersonaNatural> personas = await
CORECLIENTEContext.PersonaNatural
                .Where(x => x.CatNacionalidadId == nacionalidadID)
                .ToListAsync();
                return personas;
            }
}
```

- Luego de esto se debe agregar la direccion con su especificación de **ID** en el archivo **appsettings.json** dentro del bloque **DirectionCodes** de la siguiente manera (dichos valores podrán ser cambiados a nivel de deploy para re direccionar a otra dirección y solo se debe reiniciar el servicio correspondiente para que el cambio se haga efectivo).

```
"DirectionCodes": {
"GestorDocumenntal":"https://creditosdirectos.com.uy:48989/dmscdRest/documentos/"
},
```

Repository:

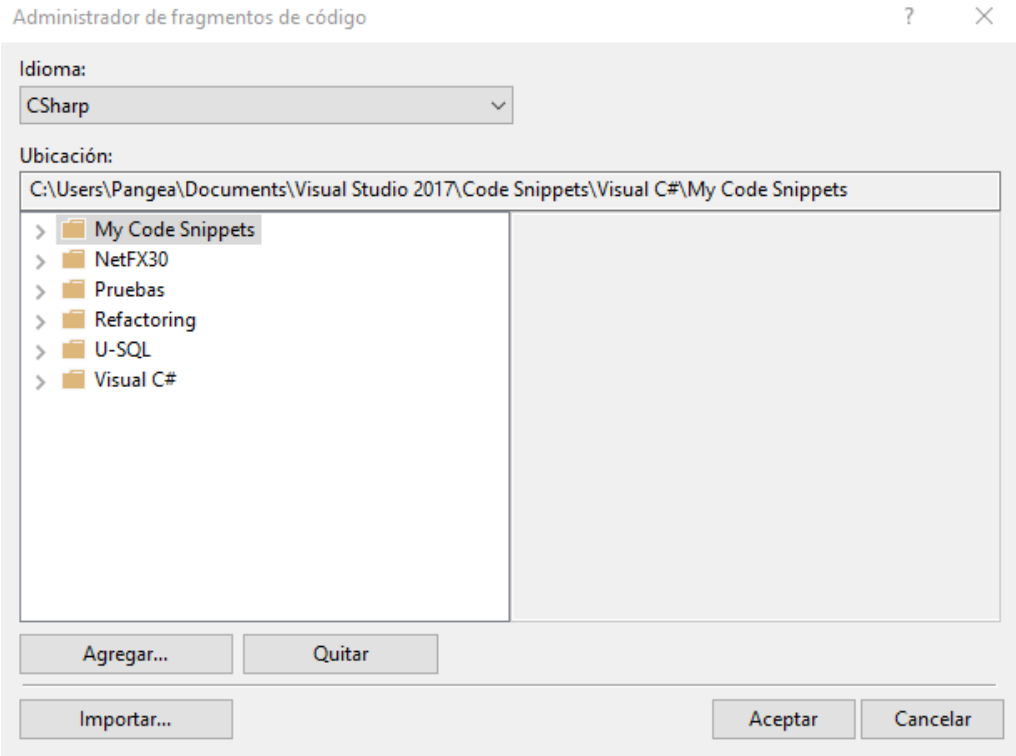
- Se encarga de manejar la clase definitoria y la interfaz **Repository** que es la encargada de controlar las operaciones comunes en todas las tablas como agregar, eliminar, encontrar y otras comunes. Dichas operaciones pueden ser usadas luego de definida la tabla en Persistence (Repository) y Core (Interfaz) en la capa API usando el UnitOfWork de la siguiente manera:

```
public async Task<CatNacionalidad> CrearNacionalidadAsync(CatNacionalidad
nacionalidad){
            _unitOfWork.CatNacionalidad.Agregar(nacionalidad);
            await _unitOfWork.Complete();
            return nacionalidad;
}

public async Task<string> EliminarNacionalidadAsync(int id){
            CatNacionalidad nacionalidad = await
_unitOfWork.CatNacionalidad.ObtenerAsync(id);
            if (nacionalidad != null)
            {
                _unitOfWork.CatNacionalidad.Eliminar(nacionalidad);
                await _unitOfWork.Complete();
                return "Nacionalidad eliminada";
            }
            else
            {
                return "Registro no encontrado";
            }
}
```

Creación de clases Repository e Interfaz

Para evitar tanta replica de código o uso de copie y pegue se generaron 2 **SNIPPETS** de los cuales el primero está diseñado para generar el código base de los repositorios y el segundo para generar el código base de las interfaces vacías (Sin métodos implementados en el REPOSITORY).

Dichos SNIPPETS son 2 archivos .snippet que deben ser importados usando el VS2017 presionando la opción HERRAMIENTAS > Administrador de fragmentos de Codigo > Importar. Allí se busca la ubicación del archivo (Ubicar los archivos en la carpeta de la solución creando una nueva carpeta llamada SNIPPETS) e importarlos de manera que puedan ser usados en cualquier momento.



El método de uso de los SNIPPETS es el siguiente en el caso de repositorio procederemos a crear la clase con el formato correspondiente, dentro de la clase borramos todo el contenido y escribimos las letras "**rep"** y presionaremos varias veces la tecla tabulación para que el código necesario se autogenere. Al generar el código algunas palabras deben ser reemplazadas como lo son el ClassName que debe ser modificado por el nombre de clase correspondiente y el ServiceName por el nombre del proyecto, dichas palabras se modificaran automáticamente solo modificando las mismas en su primera aparición en el documento. Para las interfaces el proceso es el mismo solo que el atajo son las letras **"itf"** y para las uniones el atajo son las letras **"uni".**

REPOSITORIO

```
using Microsoft.EntityFrameworkCore;
using ServiceName.Core.Models;
using ServiceName.Core.Repositories;
namespace ServiceName.Persistence.Repositories
{
    class ClassName : Repository<ClassName>
    {
        public ClassNameRepository(DbContext context) : base(context)
        { }
    }
}
```

INTERFAZ

```
using ServiceName.Core.Models;

namespace ServiceName.Core.Repositories
{
    public interface IClassNameRepositoryRepository : IRepository<IClassNameRepository>
    { }
}
```

UNION

```
using ServiceName.Api.GraphQL.QueryTypes;
using Common.Utils.QueryTypes;
using HotChocolate.Types;
namespace ServiceName.Api.GraphQL.UnionTypes
{
    public class ClassNameResultResult : UnionType
    {
        protected override void Configure(IUnionTypeDescriptor descriptor)
        {
            descriptor.Name("ClassNameResultResult");
            descriptor.Type<ClassNameResultType>();
            descriptor.Type<ErrorMessageType>();
        }
    }
}
```

Manejo de clase Startup.cs

Como previamente explicamos dicha clase se encarga de la configuración general de inicio de la app. Por lo tanto para el uso del token (Comprobación y actualización del mismo) se deben agregar en el mismo lo siguiente:

- Dentro del bloque **ConfigureServices** se agregara lo siguiente

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
                .AddJwtBearer(options =>
                {
                    options.TokenValidationParameters = new
TokenValidationParameters
                    {
                        ValidateIssuer = true,
                        ValidateAudience = true,
                        ValidateLifetime = true,
                        ValidateIssuerSigningKey = true,
                        ValidIssuer = _Configuration["Jwt:Issuer"],
                        ValidAudience = _Configuration["Jwt:Issuer"],
                        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_Configuration["Jwt:Key"]))
                    };
                });

            services.AddCors(options => {
                options.AddPolicy("CorsPolicy",
                    builder => builder.AllowAnyOrigin()
                    .AllowAnyMethod()
                    .AllowAnyHeader()
                    .WithExposedHeaders("*", "Token")
                    .Build());
            });
```

- Dentro del bloque **Configure** se agregara lo siguiente

```
app.Use(async (context, next) =>
            {
                context.Response.OnStarting(() =>
                {
                    JwtToken token = new JwtToken();
                    token.RefrescarToken(context);
                    return Task.FromResult(0);
                });
                await next();
            });
```

Dichas líneas se encargaran de refrescar el token recibido en cada solicitud de ser necesario y validar el mismo en cada llamado realizado por la aplicación usando la dirección del proyecto correspondiente la cual debe ser agregada en el archivo appsettings.json en el bloque **“DirectionCodes”** con el nombre **“GraphAuth”**.

```
"DirectionCodes": {
    "GraphAuth": "https://creditosdirectos.com.uy:48989/graph-auth/"
        },
```