



UNIVERSIDAD  
POLITECNICA  
DE VALENCIA

**Prácticas**

## *Boletín Práctica 3*

# Capa de Persistencia con Entity Framework

**Ingeniería del Software**

ETS Ingeniería Informática

DSIC – UPV

**Curso 2020-2021**

## 1. Objetivo

**IMPORTANTE:** Lea primero **todo** el documento antes de ponerse a implementar.

El objetivo de esta sesión es el desarrollo de una capa de acceso de datos que ofrezca a la lógica de la aplicación los servicios necesarios para recuperar, modificar, borrar o añadir objetos a la capa de persistencia, sin que la capa de lógica sepa qué mecanismo de persistencia particular se está usando. Para facilitar el trabajo se va a utilizar **Entity Framework (EF)**.

EF es un sistema de mapeo objeto-relacional que va a permitir a nuestra aplicación trabajar directamente con Entidades que son objetos de la lógica de negocio, y no objetos específicos para transferencia de datos. EF se encargará, automáticamente, de gestionar de forma transparente la persistencia de dichos objetos en una base de datos relacional como SQL. El acceso a datos usando Entity Framework se basa en el patrón **Repositorio + Unidad de Trabajo**, tal y como se explica en el tema 6 dedicado a la Persistencia.

Previamente a la realización de la práctica, el estudiante deberá haber repasado el “Tema 6 Diseño de Persistencia” y los seminarios asociados dedicados a “Entity Framework” y “DAL”. Por otro lado, el caso de estudio de referencia “VehicleRental”, que puede descargar de Poliformat, es una implementación completa, utilizando la arquitectura explicada en clase, que debe servir como ejemplo al estudiante en el desarrollo del caso de estudio que nos ocupa.

El objetivo de esta sesión es desarrollar las clases que forman la capa de persistencia del caso de estudio de forma análoga al ejemplo de referencia, y probar que la capa de persistencia funciona correctamente.

A continuación se detallan los distintos pasos y aspectos a tener en cuenta para implementar la capa de persistencia del proyecto.

## 2. Configuración inicial de la solución

Para trabajar con EF es necesario que añada a su proyecto el paquete necesario. Para ello, un miembro del equipo (Team Leader) hará lo siguiente: en Visual Studio vaya a Herramientas > Administrador de paquetes NuGet > Administrar paquetes NuGet para la solución y en el cuadro de texto de Examinar escriba Entity Framework. Seleccione ese paquete (ver Figura 1) y añádalo al proyecto de biblioteca de su solución (proyecto que creó en la práctica 2 que contiene las subcarpetas BusinessLogic y Persistence que se llama **GestDepLibrary**).

**Tarea 1:** Agregue el paquete EF a su proyecto siguiendo las indicaciones anteriores.

Compruebe en el Explorador de Soluciones que en las Referencias de su proyecto se incluye EF (ver Figura 2). Una vez añadido este paquete se sincronizará la solución para subir los cambios al servidor.

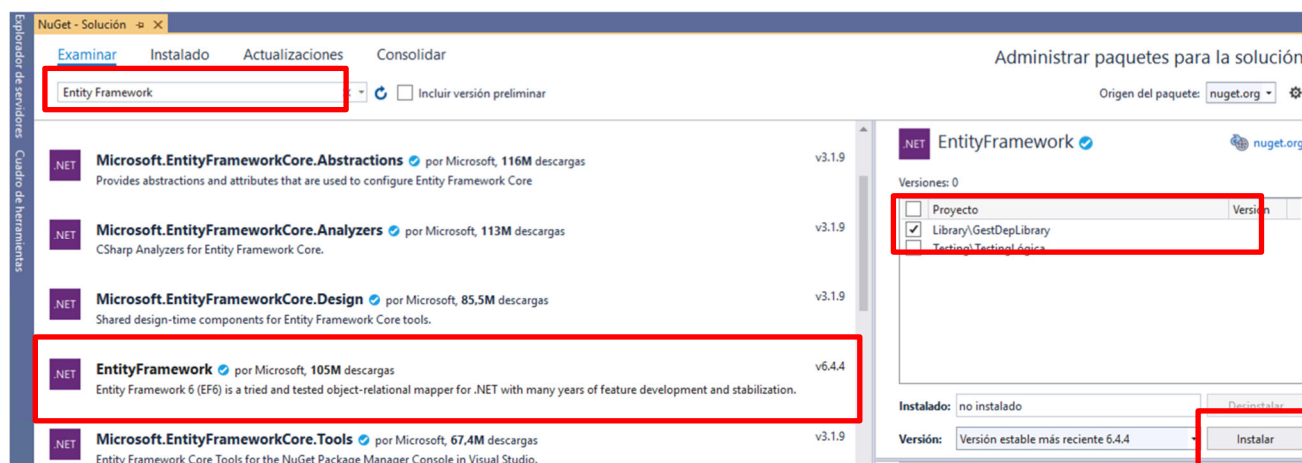


Figura 1. Añadiendo el paquete Entity Framework al proyecto de biblioteca de clases

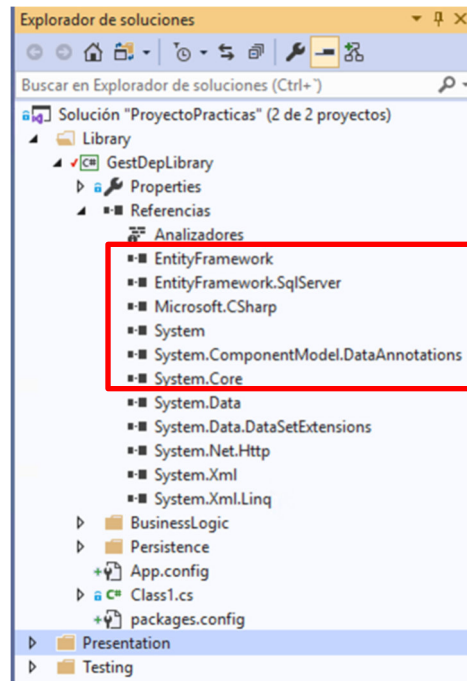


Figura 2. Comprobando que el paquete EF está instalado

### 3. Diseño de la capa de acceso a datos

La arquitectura de la capa de acceso a datos se va a diseñar de manera similar a la que se muestra en la Figura 3. Esta arquitectura es la que se utiliza en el proyecto de ejemplo VehicleRental que se proporciona. A continuación se describen los elementos de la arquitectura.

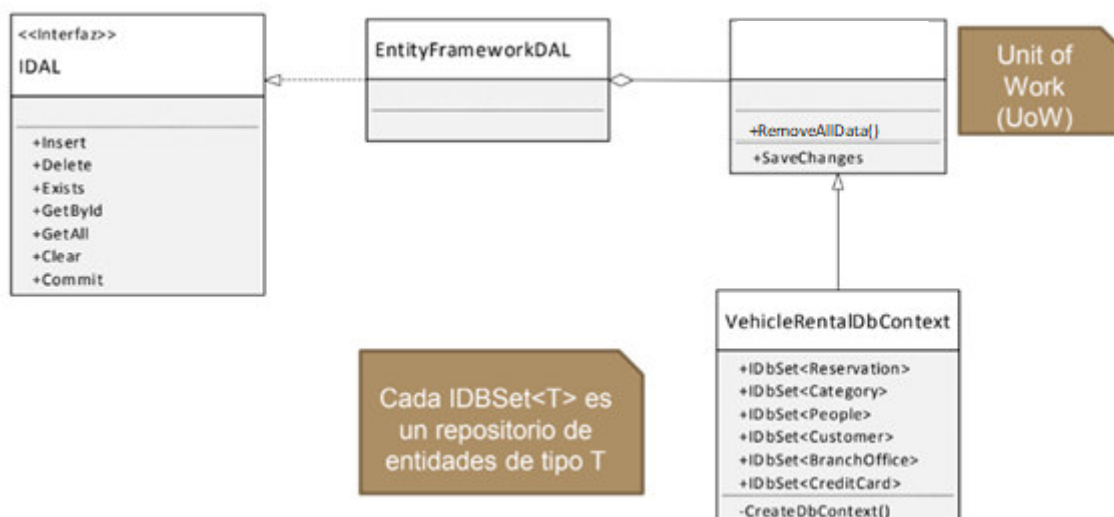


Figura 3. Arquitectura de la capa de acceso a datos

**IDAL:** La gestión transparente del acceso a datos (agnóstico respecto al mecanismo de persistencia) la vamos a lograr gracias al uso de una **interfaz** entre la capa de lógica y la capa de persistencia real. Esta interfaz, denominada **IDAL** (DAL = Data Access Layer), es la encargada de abstraer los servicios de la capa de acceso a datos del mecanismo de persistencia particular que se utilice (SQL, XML, etc.).

La Interfaz IDAL es un adaptador que combina la funcionalidad de los repositorios (Insert, Delete, GetXXX, Exists), junto con la funcionalidad de la unidad de trabajo (Commit). En efecto, si consultamos el proyecto de ejemplo VehicleRental, veremos que IDAL (archivo Persistence/IDAL.cs) tiene los siguientes métodos.

```
void Insert<T>(T entity) where T : class;
void Delete<T>(T entity) where T : class;
IEnumerable<T> GetAll<T>() where T : class;
T GetById<T>(IComparable id) where T : class;
bool Exists<T>(IComparable id) where T : class;
void Clear<T>() where T : class;
void Commit();
```

Se trata de los métodos habituales ya comentados, más un método adicional (Clear) para borrar la base de datos.

#### El uso de la genericidad:

Observe que se ha utilizado *genericidad*, reduciendo notablemente la cantidad de código a implementar. Es decir, en lugar de implementar las siguientes operaciones individuales

```
void InsertOffice(BranchOffice o);
void InsertReservation(Reservation r);
void InsertCategory(Category c);
...
```

el mecanismo de genericidad nos permite definir un único método

```
void Insert<T>(T entity) where T : class;
```

que se instanciará en ejecución en función del tipo *T* (que será una clase) correspondiente.

**EntityFrameworkDAL:** es una implementación específica de la interfaz IDAL para EF. El código de esta clase, que en el proyecto de ejemplo se encuentra en Persistence/EntityFrameworkImp/EntityFrameworkDAL.cs, es el siguiente:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Linq.Expressions;

namespace VehicleRental.Persistence
{
    public class EntityFrameworkDAL : IDAL
    {
        private readonly DbContext dbContext;

        public EntityFrameworkDAL(DbContext dbContext)
        {
            this.dbContext = dbContext;
        }

        public void Insert<T>(T entity) where T : class
        {
            dbContext.Set<T>().Add(entity);
        }

        public void Delete<T>(T entity) where T : class
        {
            dbContext.Set<T>().Remove(entity);
        }
    }
}
```

```

    }

    public IEnumerable<T> GetAll<T>() where T : class
    {
        return dbContext.Set<T>();
    }

    public T GetById<T>(IComparable id) where T : class
    {
        return dbContext.Set<T>().Find(id);
    }

    public bool Exists<T>(IComparable id) where T : class
    {
        return dbContext.Set<T>().Find(id) != null;
    }

    public void Clear<T>() where T : class
    {
        dbContext.Set<T>().RemoveRange(dbContext.Set<T>());
    }

    public void Commit()
    {
        dbContext.SaveChanges();
    }

    public IEnumerable<T> GetWhere<T>(Expression<Func<T, bool>> predicate) where T :
class
    {
        return dbContext.Set<T>().Where(predicate).AsEnumerable();
    }
}

```

Se observa que la clase DAL contiene una referencia a un objeto DbContext, tal y como se ilustra en la Figura 3, y la implementación particular de cada uno de los servicios que ofrece la interfaz IDAL. DbContext define el mapeo de objetos del dominio a tablas de la base de datos, siguiendo el patrón Repositorio + UoW.

**VehicleRentalDbContext:** esta clase extiende DbContext y en ella se definen las propiedades que implementan los repositorios para persistir las clases del negocio, y algunas opciones de configuración y creación de la base de datos.

En el proyecto de ejemplo si observa la clase VehicleRentalDbContext, que se encuentra en Persistence/EntityFrameworkImp/VehicleRentalDbContext.cs, observará las siguientes características:

- Hereda de DbContext
- Tiene propiedades de tipo IDbSet<Tipo> donde “Tipo” es cada una de las clases del modelo que han de ser persistentes en la base de datos (cada IDbSet constituye un repositorio con los métodos típicos para acceder, añadir o eliminar objetos). En concreto, en el proyecto de ejemplo tenemos los siguientes

```

public IDbSet<BranchOffice> BranchOffices { get; set; }
public IDbSet<Reservation> Reservations { get; set; }
public IDbSet<Category> Categories { get; set; }
public IDbSet<Person> People { get; set; }
public IDbSet<Customer> Customers { get; set; }
public IDbSet<CreditCard> CreditCards { get; set; }

```

- Tiene un constructor que proporciona al constructor base el nombre de la cadena de conexión a la base de datos ("VehicleRentalDBConnection") que está definida en el archivo App.config del proyecto de inicio (NO de la biblioteca de enlace dinámico). En el constructor se incluyen también algunos parámetros de configuración.

```
public VehicleRentalDbContext() : base("Name=VehicleRentalDbConnection") { ... }
```

Para implementar la capa de persistencia del proyecto del caso de estudio, tenga en cuenta lo siguiente:

- En PoliformaT encontrará un archivo EntityFrameworkImp.zip que contiene la implementación de las clases DbContextISW, EntityFrameworkDAL e IDAL.
- La implementación de las clases IDAL y EntityFrameworkDAL es genérica y se pueden reutilizar en el proyecto. La clase DbContextISW es necesaria para implementar la clase unidad de trabajo del proyecto: GestDepDbContext.
- Deberá crear una estructura como la de la Figura 4, agregando la carpeta EntityFrameworkImp en Persistence y los archivos DbContext.cs, EntityFrameworkDal.cs e IDAL.cs.
- Para añadir un archivo al proyecto, pulse con el botón derecho del ratón sobre la carpeta donde quiera incluirlo y seleccione Agregar > Elemento Existente .... A continuación, seleccione el archivo o archivos .cs que quiera añadir.
- La unidad de trabajo, clase **GestDepDbContext**, se implementará extendiendo la clase DbContextISW, que a su vez extiende la clase DbContext de EF, que proporciona un método llamado RemoveAllData() que implementa el borrado de la BD.
- Para implementar la clase GestDepDbContext, fíjese en la implementación de VehicleRentalDbContext y hágala de forma similar, definiendo una propiedad IDbSet para cada una de las clases persistentes del modelo. Por ejemplo:

```
public IDbSet<CityHall> CityHalls { get; set; }
```

Como las clases del modelo están definidas en un namespace diferente (GestDep.Entities) deberá incluir la correspondiente directiva using GestDep.Entities.

- Tendrá que asignar todas las clases creadas al mismo namespace: GestDep.Persistence.

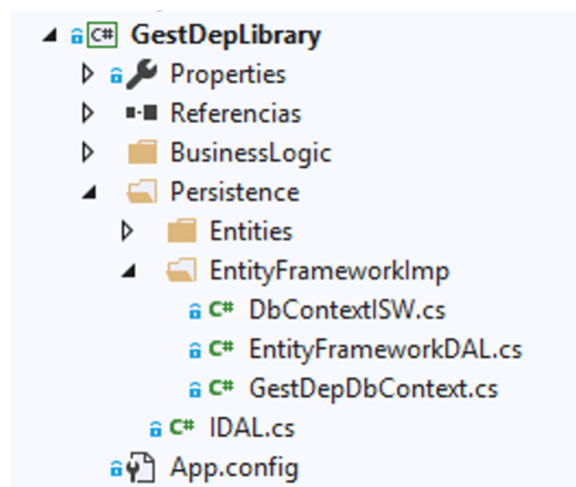


Figura 4. Estructura de la capa de persistencia en el proyecto de VS

**Tarea 2:** Implementar la capa de persistencia para el proyecto, siguiendo las consideraciones anteriores y tomando como referencia el proyecto de ejemplo.

#### 4. Utilización de Data Annotations

A veces es necesario modificar el código de la capa lógica mediante las **Data Annotations** para proporcionar cierta información a EF. En el seminario 6.1 de teoría sobre **Entity Framework** se dan algunas pautas en relación al uso de anotaciones. Concretamente, preste atención a las anotaciones:

```
[Key]
[InverseProperty("XXX")]
```

**Tarea 3:** Revise el código de la capa lógica y añada las anotaciones EF que considere necesarias.

#### 5. Modificación de los constructores

En la práctica anterior definió los constructores necesarios para crear los objetos de la lógica de negocio. Siguiendo las instrucciones, definió un constructor por defecto y un constructor con argumentos.

El constructor por defecto lo utiliza EF cuando tiene que materializar un objeto en memoria después de recuperar la información de las tablas de la BD. Por eso no tiene que inicializar los atributos del objeto (ya lo hace EF) pero sí hay que inicializar los atributos de tipo colección.

El otro constructor con argumentos es el que se debe utilizar en el código del programa cuando sea necesario crear un objeto. Como ya sabrá, EF se encarga de dar valor a los atributos ID de tipo numérico de manera automática en el momento en el que se persiste el objeto por primera vez. Por lo tanto, no es necesario dar valor a dichos atributos en el constructor del objeto ya que EF le cambiará el valor en cuando lo persista por primera vez. Los constructores ya se diseñaron en la sesión anterior teniendo en cuenta este aspecto. Únicamente los ID de tipo string deberán ser inicializados en el constructor.

**Tarea 4:** Revise las clases de la lógica y compruebe que los constructores están correctamente implementados según se indica en el párrafo anterior.

#### 6. Prueba de la capa de persistencia (proyecto de consola)

Una vez haya finalizado el desarrollo de la capa de persistencia sincronice su solución con un comentario como "Capa Persistencia Finalizada (versión beta)".

Es el momento de probar la capa de persistencia. Para ello puede crear un proyecto de consola, tal y como se vio en la primera sesión de prácticas, y escribir un programa para crear algunos objetos de la lógica de negocio y persistirlos usando los servicios ofrecidos por el DAL. Los pasos a seguir serían:

- Cree una carpeta de soluciones llamada **Lab3** dentro de **Testing** con un proyecto de consola llamado DBTest.
- Incluya una referencia a su biblioteca de clases (GestDepLibrary). Para ello, en su proyecto de aplicación de consola (DBTest): Botón derecho del ratón sobre Referencias > Agregar referencia ... y seleccionar dentro de Proyectos su proyecto de biblioteca de clases.
- Agregue el paquete Entity Framework usando el administrador de paquetes NuGet, de forma similar a como se ha explicado en el apartado 2.
- Modifique el archivo de configuración App.config que se habrá creado en DBTest, añadiendo la cadena de conexión a base de datos. Si usamos SQLServer como motor de base de datos y denominamos "GestDepDB" a la base de datos a crear, la cadena de conexión que debe añadir es ésta:

```
<connectionStrings>
  <clear />
```

```
<add name="GestDepDBConnection"
connectionString="Server=(localdb)\mssqllocaldb;Database=GestDepDB;Trusted_Connection
=True;MultipleActiveResultSets=true" providerName="System.Data.SqlClient" />
</connectionStrings>
```

En el código de prueba (implementado en Program.cs) habrá que crear en primer lugar una instancia de EntityFrameworkDAL pasándole una instancia de GestDepDbContext. Al crear la instancia de GestDepDbContext por primera vez se creará la base de datos. A continuación, deberá crear algunos objetos de la lógica de negocio y persistirlos usando los servicios ofrecidos por el DAL.

Para facilitar el trabajo y asegurar que la base de datos se crea correctamente, le proporcionamos el archivo Program.cs con parte del código necesario. Este archivo deberá agregarlo al proyecto DBTest. La estructura resultante del proyecto será como la de la Figura 5:

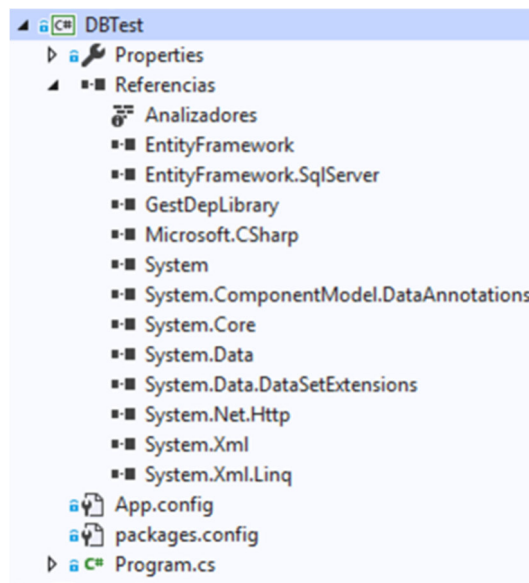


Figura 5. Estructura del proyecto de consola de Test

Añada este proyecto (DBTest) a la solución y abra el archivo Program.cs. Observará un método prácticamente vacío denominado createSampleDB que habrá que rellenar con el código necesario para crear los objetos de negocio y persistirlos en la base de datos.

```
private void createSampleDB(IDAL dal)
{
    // Remove all data from DB
    dal.RemoveAllData();

    CityHall ch = new CityHall("Valencia");
    dal.Insert<CityHall>(ch);
    dal.Commit();

    Gym g = new Gym(Convert.ToDateTime("14:00:00"), 5, 5, 2.00, "Gym1",
        Convert.ToDateTime("08:00:00"), 46022);
    dal.Insert<Gym>(g);
    ch.Gyms.Add(g);
    dal.Commit();
}
```



```
// Populate here the rest of the database with data  
}
```

Como se puede observar en el código después de borrar el contenido de la base de datos (`dal.RemoveAllData()`), se crean una instancia de `CityHall`. Para que los datos persistan en la base de datos debemos añadir el objeto al repositorio de `CityHalls`, con la instrucción `dal.Insert<CityHall>(ch)`, y a continuación realizar el commit en la base de datos.

De forma similar creamos un objeto `Gym`. En este caso, además habrá que mantener la relación entre `CityHall` y `Gym`, agregando el objeto `Gym` a la colección `Gyms` de `CityHall` (`ch.Gyms.Add(g)`).

Para poder ejecutar este proyecto deberá indicar que es el proyecto de inicio. Para ello, deberá pulsar con el botón derecho del ratón en el proyecto y en el menú contextual escoger la opción “Establecer como proyecto de inicio”.

Es importante recordar que tras completar una transacción (una o más operaciones que implican un cambio en los datos), debe invocarse al método `Commit` para persistir todos los cambios. Aunque no se ofrece un método específico para actualizar un objeto, **si se han modificado objetos internamente, igualmente tendremos que ejecutar el método `Commit`**.

**Tarea 5:** Cree un programa de prueba siguiendo las indicaciones anteriores que persista en la base de datos una instancia de cada una de las clases del modelo. Para comprobar si la información se ha almacenado en la base de datos vea el siguiente apartado.

## 7. Herramientas de Inspección de la BD

Desde el propio entorno de desarrollo en Visual Studio es posible conectarse a cualquier base de datos para ver sus tablas y su contenido. Para conectarse a una base de datos existente local (creada como un fichero local) tendrá que realizar los siguientes pasos: Herramientas > Conectar con la Base de Datos y seleccionar como origen de datos “Archivo de base de datos de Microsoft SQL Server (SqlClient)” y como archivo de datos seleccionar el fichero con extensión “.mdf” creado. Dada la configuración realizada en el fichero `App.config`, el fichero de base de datos “.mdf” se crea en `C:\Users\nombreUsuario\GestDepDB.mdf`.

Una vez creada la conexión es posible explorar las tablas de la base de datos en el explorador de servidores que aparecerá en el entorno de desarrollo, como muestra la Figura 6.

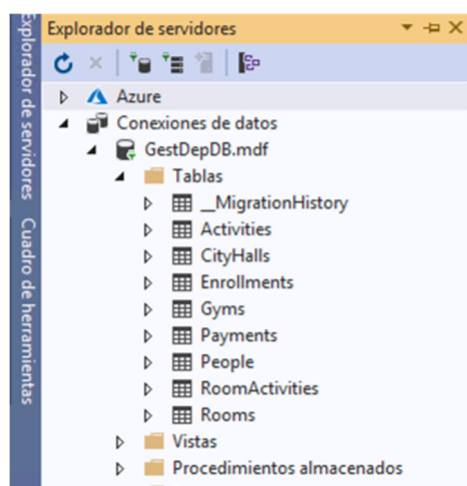
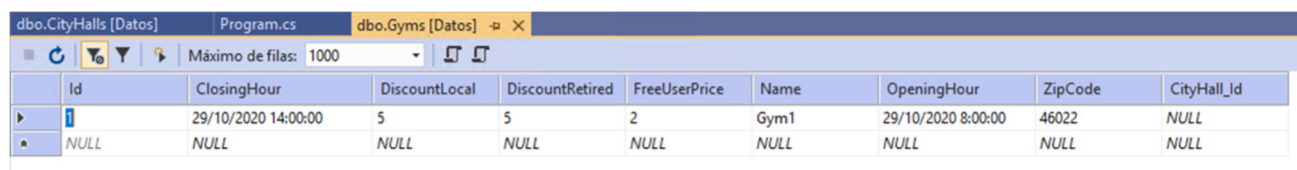


Figura 6. Tablas de la BD desde el explorador de servidores

Haciendo doble-click sobre una tabla se puede ver su estructura. Para ver los datos de una tabla se hará click sobre el botón derecho de ratón sobre la tabla > Mostrar datos tabla. Por ejemplo, en la tabla Gym, después de ejecutar el programa de prueba, veremos lo siguiente:



	Id	ClosingHour	DiscountLocal	DiscountRetired	FreeUserPrice	Name	OpeningHour	ZipCode	CityHall_Id
		29/10/2020 14:00:00	5	5	2	Gym1	29/10/2020 8:00:00	46022	NULL
	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figura 7. Contenido de la tabla Gym

De forma similar, podemos elegir la opción Nueva consulta que nos permitirá escribir sentencias SQL que trabajen sobre nuestra base de datos. Por ejemplo, fácilmente podremos visualizar todos los registros que hay en nuestra en una tabla TTT de la base de datos, más concretamente en la tabla denominada TTT, escribiendo “Select \* From TTT” y ejecutando la consulta (botón triángulo verde “play” del SQLQuery creado).

**Tarea 6:** Compruebe en la base de datos que se ha almacenado la información en cada una de las tablas.

**IMPORTANTE:** Durante la realización de estas actividades, sincronice su solución cuando lo considere conveniente y añada comentarios informativos de los cambios introducidos. Una vez validado que el programa funciona correctamente realice el correspondiente commit.

## 8. Pruebas unitarias (MSTest)

Las pruebas de código pueden sistematizarse mejor mediante un motor de pruebas unitarias como MSTest<sup>1</sup>. En Visual Estudio pueden definirse casos de prueba para garantizar la cobertura del código y detectar errores en el mismo. VS proporciona herramientas para la ejecución sistemática de las pruebas unitarias y la visualización del resultado de las mismas.

**Tarea 7.** Como última tarea antes de pasar a la implementación de la lógica de los casos de uso, deberá ejecutar las pruebas unitarias que le indique su profesor o profesora de prácticas.

<sup>1</sup> Puede informarse sobre pruebas unitarias en VS en:

<https://docs.microsoft.com/es-es/visualstudio/test/getting-started-with-unit-testing?view=vs-2019>