TÉCNICAS DE PERSISTENCIA DE DATOS CON .NET Y PROGRAMAS ERP

Producto 2

MIEMBROS

Marc Planas Chamorro
Daniel Benítez Nadal
Daniel Zafra del Pino
Daniel Boj Cobos

Grupo

HYPERION

Contenido

Producto 2. Implementación de la capa de acceso a datos de la aplicación GenteFit

Enlace a GitHub

Para realizar el seguimiento de las tareas que vamos realizando, tanto a nivel individual como grupal, compartimos el enlace a nuestro repositorio de GitHub:
https://github.com/DanielBoj/UOC_Persistencia_con_NET_y_ERP.git

Enlace a video-demostración en YouTube

https://youtu.be/yU_esu9VBJU

Conexión entre una APP desarrollada con .NET y una BBDD

Para empezar, aunque resulte evidente, **hay que entender que el backend de nuestra aplicación se va a escribir en un lenguaje distinto del que usa nuestra BBDD, incluso con un paradigma distinto**. Nuestra app va a estar escrita en un lenguaje de .NET (C#, F# o VisualBasic), mientras que nuestra base de datos, partamos de momento de la premisa de que va a ser relacional, se implementará mediante *querys* escritas en SQL. De este modo, para poder tener acceso a nuestra BBDD desde nuestra aplicación, **tendríamos que mezclar lenguajes y principios muy distintos**. Así, para poder lanzar las consultas, se parte de la base de que es necesario importar los conceptos de nuestra base de datos a los lenguajes con los que vamos a implementar nuestro *backend* y, además, implementar instrucciones escritas en SQL dentro de nuestro código a través de strings o de objetos especializados.

Por otro lado, también tenemos **marcadas diferencias entre las dos herramientas al momento de implementar el manejo de los datos**. Es decir, los tipos de datos pueden diferir marcadamente tanto en concepto como en implementación y tratamiento entre las distintas plataformas que escojamos. Además, las asociaciones entre objetos pueden ser unidireccionales, mientras las asociaciones relacionales siempre son bidireccionales.

Por último, vamos a encontrar una diferencia muy importante entre nuestra app y la BBDD, ya que, como comentábamos, sus paradigmas son muy distintos. **En nuestro backend, estamos usando la Programación Orientada a Objectos**, con lo cual, el acceso y manejo de los datos mediante las clases y su instanciación nos va a permitir cosas como incluir otros objetos como atributos de nuestros objetos, relacionándolos entre ellos a través de la instanciación directamente. En cambio, **en las bases de datos relacionales, esto resulta impensable, a diferencia, tendremos múltiples tablas que deberemos relacionar a través de Primary Keys y Foreigns Keys**. Con ello, deberemos adaptar la forma de trabajar de nuestro *backend* para poder manejar la lógica que aplicamos en nuestra base de datos relacional.

Por todos estos motivos, podemos deducir que **será muy dificultoso mantener un flujo de información estable entre una app y una base de datos SQL**, apareciendo un efecto que se conoce como *desfase de* impedencia, teniendo que basarnos, de salida, en bibliotecas de funciones que adaptan el lenguaje POO a los paradigmas SQL, **forzándonos a trabajar con los conceptos de la BBDD y saliendo de la metodología de POO.**

Para tratar de mantener los principios de Orientación a Objetos en el acceso a las BBDD relacionales, **surgieron las bibliotecas especializadas y los *frameworks***.

ORM

Para permitir mantener la abstracción característica de la POO, pueden usarse **bibliotecas especializadas en el acceso a datos que se encargan de generar la lógica para interactuar con las BBDD mediante el mapeo de objetos a lenguaje relacional y al contrario**, de ahí sus siglas, *Object-Relational-Mapper.* De este modo, usamos los métodos y clases ofrecidos por la biblioteca ORM para no tener que usar lógica relacional y tener que escribir *querys* en SQL. La biblioteca se encarga de implementar toda la lógica necesaria para generar tablas a partir de clases o clases a partir de las tablas y sus relaciones.

Esto es una gran ventaja ya que usando una biblioteca ORM **no tenemos que preocuparnos para nada en realizar consultas SQL**, podemos administrar la BBDD a través de la lógica de nuestra app si tener que acceder a la BBDD de manera directa.

Por consiguiente, podemos enumerar una serie de ventajas esenciales al usar un ORM para implementar la interactuación con las BBDD relacionales:

- Eliminan la necesidad de escribir código SQL.
- Mantienen los principios de la POO, especialmente de la herencia entre clases.
- Permiten aumentar la modularización y reusabilidad del código. Además, mejoran la inteligibilidad, facilitan el mantenimiento y la escalabilidad de la aplicación; manteniendo siempre la independencia entre la aplicación y la base de datos.
- Evitan los ataques por inyección SQL.
- Evitan tener que crear la lógica interna de conexión con la BBDD.

Frameworks ORM para .NET

Tanto Entity como NHibernate basan la persistencia de datos y la comunicación entre los dos modelos en la **traducción de los datos de las BBDD relacionales a XML**.

Entity Framework

Es el ORM más usado y conocido para trabajar con BBDD SQL en un entorno .NET y ha sido creado por Microsoft, la misma compañía que creó y mantiene .NET. Con ello, sabemos que **siempre va a gozar de la máxima compatibilidad con la plataforma de desarrollo** y que ahora mismo se encuenta en su sexta versión y cuenta con un fork llamado EF Core, más moderno, extensible y ligero.

Como hemos dicho, EF nos va a permitir escribir las aplicaciones en lenguajes de .NET que pueden acceder e interactuar fácilmente y sin abandonar el paradigma de Orientación a Objetos con bases de datos relacionales adaptándolos a los objetos fuertemente tipados que vamos a desarrollar en la aplicación. Ahorrándonos, además, que tengamos que realizar toda la implementación lógica para conseguirlo.

NHibernate

Es un proyecto ORM Open Source que realiza la **migración del *framework* Hibernate de Java a un entorno .NET**. Se construye sobre ADO.NET, una biblioteca de clases pensadas para desarrollar el acceso a datos de las aplicaciones .NET, y es una opción muy apreciada y querida por muchos desarrolladores ofreciendo una base de código madura, estable y con la contribución de multitud de programadores para mantenerla altamente activa.

Como ORM, nos ofrece un marco para poder realizar el mapeo entre el dominio orientado a objetos con objetos fuertemente tipados de nuestra app y el modelo relacional de la BBDD.

Otros frameworks

Además de estas dos opciones, que son las más populares, también existen otros ORM como Dapper, que es mucho más sencillo y se implementa en escenarios donde no haga falta una gran complejidad y Subsonic, que, a diferencia del caso anterior, es un mapeador muy complejo y con una curva de aprendizaje muy pronunciada.

Otros lenguajes de programación

Java

En Java, el ORM más usado y conocido es Hibernate, que como hemos visto, tiene una migración a .NET. Este *framework* es gratuito y Open Source y está mantenido por Red Hat, signo de calidad. Hibernate se basa en JDBC, la API de Java para la conexión con Bases de Datos, y a la vez, implementa su propia API para gestionar las peticiones a y las respuestas de la BBDD. Igual mente utiliza JPA, la API de persistencia de Java, que es una especificación que define la persistencia de datos mediante Java y se basa en XML. Así mismo, Hibernate permite aplicar fácilmente los principios de herencia, polimorfismo, asociación y composición y las Colecciones de Java manteniendo un alto rendimiento y asegurando la escalabilidad y extensibilidad de los proyectos.

Otros frameworks

Otras opciones presentes en Java son: Jooq, ActiveJDBC y QueryDSL.

Python

Django es el principal y más conocido *framework* para el desarrollo web mediante Python e incluye una biblioteca ORM propia. Así, en este caso, nos encontramos con un *framework*

completo para el desarrollo web que nos ofrece de forma nativa todas las funcionalidades necesarias para implementar la persistencia de datos a través del acceso e interacción con BBDD relacionales. Una de las principales características de Django es su notable velocidad y un paradigma especialmente enfocado a la seguridad, ayudando a los desarrolladoras a evitar la mayoría de los errores de seguridad que se cometen en el desarrollo web.

Otros frameworks

También en Python tenemos otras opciones como: Peewee y Pony ORM, además, este lenguaje presume de uno de los ORM con mejor fama entre los programadores, SQLAlchemy.

Inconvenientes y porqué hemos optado por otra opción

Durante mucho tiempo, las bases de datos relacionales fueron la elección de cabecera. Pero este escenario lleva ya unos años cambiando con la **popularización de bases de datos no-relacionales**. Uno de los escenarios donde más se ha visto este crecimiento es el del desarrollo de aplicaciones web, sobre todo desde la normalización de aplicaciones SPA. **Los ORM, al final, responden y reflejan la complejidad de las bases de datos relacionales y acaban siendo bibliotecas complejas y densas**. Así, podemos encontrar varios inconvenientes en el uso de *frameworks* ORM y, por ende, de bases de datos relacionales para implementar la permanencia de datos de nuestra app:

- Alta complejidad de las bibliotecas OMR, con una curva de aprendizaje muy pronunciada.
- Añaden una capa pesada de complejidad a nuestro código, haciendo que muchas veces empeore su rendimiento. Esto sucede sobre todo en casos en los que anidemos varias consultas y, además, debemos tener en cuenta que las bases de datos relacionales pueden ser especialmente lentas comparadas con otras opciones.
- La configuración inicial de nuestra aplicación puede escalar de forma desproporcionada al aumentar el número de entidades o su complejidad.
- Dependencia del desarrollo del modelo relacional de la BBDD para entender bien su funcionamiento y administrarla y gestionarla de forma más eficiente.
- Las BBDD relacionales, además de las funciones CRUD, suelen contar con tareas, *triggers* y funciones internas que pueden ser complicadas de traducir en nuestra app.

Ya que **vamos a desarrollar una app web enfocada a mobile first** y queremos trabajar con un escenario lo más real posible, **hemos sopesado todas las opciones y hemos decidido no trabajar con una BBDD SQL ni un ORM. Así, hemos optado por usar MongoDB, la base de datos no-SQL más popular actualmente y una de la más usadas de forma global**, teniendo en cuenta opciones relacionales y no-relacionales, y su framework para ASP.NET, MongoDB.Driver.

MongoDB se basa en Colecciones de objetos conocidos como Documentos **implementados a través de una versión especial de JSON conocida como BSON**. Esto no es una cuestión baladí, este modelo permite que sea mucho más sencillo traducir los principios de la POO a nuestra BBDD, ya que trabaja igualmente con objetos que pueden contener otros objetos o colecciones de objetos y que **suprime toda la lógica de las relaciones bidireccionales**. Así,

se pueden traducir de forma directa todos los principios de la POO. Además, el tipo de datos se reduce a las cadenas de texto que formarán nuestros objetos JSON.

Otros aspectos que hemos tenido en cuenta y por los que se escoge MongoDB son:

- **Alto rendimiento:** Al estar basado en objetos JSON, el acceso y lectura a los datos en MongoDB es altísimo y está especialmente preparado para situaciones de alta concurrencia de solicitudes. Si bien la escritura puede ser algo más lenta, en situaciones donde predomina la lectura, como es en las aplicaciones web, el rendimiento de esta base de datos está a años luz de una relacional.
- **Ligereza:** De igual forma, como los objetos JSON son simplemente cadenas de texto más o menos largas y anidadas, son objetos muy ligeros. Además, al no tener que implementar la lógica relacional y la indexación pesada de las bases SQL, nuestra base de datos va a ser mucho más ligera que una relacional.
- **Poco consumo de recursos:** MongoDB puede levantarse en servidores muy básicos, cosa que puede suponer un tremendo ahorro para llevar a producción aplicaciones pequeñas. Además, dispone de Atlas, un cloud donde tenemos 500MB para poder implementar nuestras BBDD en la nube de forma gratuita.
- **Es gratuita:** MongoDB es una herramienta Open Source y gratuita donde solo se paga si se necesita soporte.
- **Documentación y actividad:** Como con todos los ejemplos de software Open Source, goza de una maravillosa documentación y una gran comunidad a sus espaldas.
- **Está muy adaptada a los lenguajes de desarrollo web y el *backend*.**

Tecnologías para nuestra aplicación

Usaremos un modelo **MVC + API** para implementar la aplicación ya que permite separar la lógica de negocio de la implementación del *frontend*. Nuestro controlador principal funcionará como API para gestionar los *requests* y *responses* que quedará aislado de los modelos de datos y del *frontend*, que vamos a realizar implementando una app Angular que consuma nuestra API.

Dentro de .NET, usaremos ASP para crear una aplicación web que se conecte a nuestra BBDD para recuperar los datos necesarios e interactuar con ellos y los ponga a disposición de nuestro *frontend*. **ASP.NET** es un *framework* de desarrollo web para .NET Open Source y creado por Microsoft que extiende .NET y lo prepara para realizar un desarrollo web de forma mucho más sencilla y potente. Además, como comentamos en el producto anterior, crearemos clases genéricas para modelizar las colecciones que vamos a usar realizando una generalización sobre las estructuras de datos presentes en las bibliotecas estándar del *framework* .NET, BCL. Como lenguaje de programación, usaremos **C#** para el *backend* y, al desarrollar el *frontend* en AngularJS, usaremos HTML, SASS y **Typescript.** De ASP.NET también aprovecharemos sus bibliotecas para manejar la autenticación y encriptado.

Como base de datos, usaremos una del tipo NoSQL ya que para esta app podemos aprovechar las ventajas y la flexibilidad que nos ofrecen estas BBDD porque no necesitamos una BBDD relacional con una estructura tan sólida como las SQL. Hemos escogido **MongoDB** ya que es la más usada ahora mismo en el desarrollo web. Basa su funcionamiento en *colecciones*, que se pueden entender como las tablas tradicionales, donde se guardan objetos en forma de JSON que se conocen como *documentos*, pero huyendo de cualquier tipo de relación, las colecciones

establecen los modelos de datos a través de una serie de atributos que deberán tener los objetos de estas. Una de las mayores ventajas de MongoDB, es que todos sus objetos se pueden generar directamente desde las llamadas de la API, hasta las BBDD, y además, al ser objetos JSON, no necesitamos intermediar con modelos XML. Es decir, cuando nuestra API se conecte por primera vez a nuestro clúster de MongoDB, creará la BBDD y, cada vez que realicemos una llamada HTTP del tipo POST, creará la colección o documento que necesitemos. Además, podemos crear toda la lógica de los modelos directamente en nuestra app, sin necesidad de prototiparlo previamente en la BBDD, toda la información se transmitirá a la BBDD a través de las llamadas de la API.

Para monitorizar los cambios en MongoDB y poder actuar directamente sobre la BBDD, usaremos el interfaz visual **MongoDB Compass**. Además, para testar que las llamadas de la API se realizan de forma correcta, usaremos **Postman.**

Por último, como hemos comentado, ya que hemos decidido realizar una aplicación web que, además, resulte fácil de migrar a una aplicación nativa para móvil, usaremos **AngularJS** para generar el *frontend*.

Tipo de Proyecto

En Visual Studio podemos preparar toda la estructura de proyecto escogiendo crear un proyecto del tipo: Aplicación web ASP.NET Core (Modelo-Vista-Controlador). Con ello, el mismo IDE se encargará de inicializar un proyecto con toda la estructura de directorio necesaria para empezar a trabajar en nuestra app.

Por otro lado, cuando comencemos a trabajar en el *frontend*, tras haber implementado la lógica del *backend*, deberemos crear un proyecto Angular dentro del directorio Vista. Para ello, instalaremos y usaremos Angular/Cli, que nos permite ejecutar los comandos de angular de forma sencilla desde un terminal.

Iniciando el proyecto en Visual Studio

Directorios

wwwroot → Archivos estáticos: CSS y Scripts.

Controller → Package para implementar el módulo con el controlador principal.

Model → Package para implementar el módulo con la lógica de negocio y los modelos de datos.

Views → Package para implementar el frontend, en nuestro caso crearemos una app Angular.

Capa de acceso a datos – Implementación de las acciones CRUD

A continuación, vamos a mostrar cómo hemos implementado la lógica necesaria para que nuestra aplicación sea capaz de acceder e interactuar con la base de datos MongoDB.

Acceso al servidor y a la Base de Datos

Siguiendo el modelo de clases del producto 1, la clase DataConnection implementa toda la lógica necesaria para realizar la conexi\on a la BBDD. Para ello, usa las clases de la biblioteca creada por la propia MongoDB, **MongoDb.Driver** que hemos instalado en nuestro Proyecto a través de los paquetes NuGet. Dentro de esta clase se crean el cliente de acceso al servidor y la base de datos, los procesos se ejecutarán al instanciarla. Para realizar la conexión necesitamos indicar un string de conexión que nos facilita el cluster de MongoDB en Atlas y otro string donde indiquemos el nombre de la BBDD que vamos a crear.

```
using MongoDB.Driver;

namespace GenteFit.Models.Repositories
{
    17 referencias
    public class DataConnection
    {
        // Cliente para la conexión al servidor MongoDB
        public MongoClient client;

        // Interfaz de referencia a la BBDD
        public IMongoDatabase db;

        // Creamos el constructor
        8 referencias
        public DataConnection()
        {
            // Inicializamos el cliente y conectamos con el servidor Atlas -> Obtenemos el String de conexión comc
            client = new MongoClient(env.Development.mongo_db.mongo_db_url);

            // Si Mongo no encuentra la BBDD en el servidor, la creará
            db = client.GetDatabase(env.Development.mongo_db.mongo_db_name);
        }
    }
}
```

Creación de los modelos de colecciones

Para poder comunicarnos con nuestra BBDD debemos traducir de alguna forma las clases que implementemos en nuestra aplicación a las Colecciones que usa MongoDB, por suerte, la biblioteca de MongoDB se encargará de realizar este paso mediante la clase IMongoCollection que recibe una de nuestras clases nativas y la convierte en una Collection de MongoDD. De este modo, temenos que crear los modelos de colecciones para cada una de nuestras entidades. Dentro de cada una de ellas, instanciaremos nuestra clase DataConnection, generaremos la conexión con la BBDD e implementaremos la lógica con los métodos para realizar las acciones CRUD mediante los verbos de acciones de MongoDB, teniendo en cuentas, además, que debemos usar métodos asíncronos para manejar los *request* a nuestra BBDD de tipo cloud.

A continuación, se muestra el encabezado de uno de los modelos:

```
using MongoDB.Driver;
using GenteFit.Models.Repositories.Interfaces;
using MongoDB.Bson;

namespace GenteFit.Models.Repositories.Collections
{
    2 referencias
    public class CentroCollection : ICentro
    {
        // Referencia al repositorio de datos de MongoDB e inicializamos la instancia, así nuestra clase irá a bus
        internal DataConnection _repository = new();

        // Importamos el Driver y nuestro modelo de referencia.
        private IMongoCollection<Centro> Collection;

        // Si la colecciónb no existe, Mongo la creará automáticamente.
        1 referencia
        public CentroCollection()
        {
            // Enlazamos nuestro modelo con la colección en MongoDB indicando nuestro modelo de referencia y tenie
            Collection = _repository.db.GetCollection<Centro>("Centro");
        }

        2 referencias
        public List<Centro> GetAllCentros()
        {
```

Interfaces

Para estructurar todo el conjunto de métodos que usarán nuestros modelos y para poder manejarlo desde los controladores que funcionarán como *middlewares*, creamos también las interfaces para los modelos de colecciones.

```
using GenteFit.Models;
using GenteFit.Models.Collections;

namespace GenteFit.Models.Repositories.Interfaces
{
    2 referencias
    public interface ICentro
    {
        2 referencias
        bool InsertCentro(Centro centro);
        2 referencias
        bool UpdateCentro(Centro centro);
        2 referencias
        bool DeleteCentro(string id);
        2 referencias
        List<Centro> GetAllCentros();
        4 referencias
        Centro GetCentroById(string id);
    }
}
```

Controladores

En esta fase del desarrollo, hemos realizado el test de las funciones CRUD implementando un *frontend* temporal mediante Vistas Razor. Presentan la ventaja de que se generan rápidamente con scaffolding y pueden personalizarse rápidamente para adaptarlas a las particularidades de nuestros modelos. Para manejar la comunicación entre nuestra app y la BBDD y las vistas del *frontend*, necesitamos generar los controladores que ejerzan de puente entre nuestras herramientas.

Los controladores heredan de la clase Controller e instancian la colección a través de la clase Interface de cada uno de nuestros modelos. A continuación, generan la lógica necesaria para conectar las peticiones a la BBDD con nuestras vistas de *frontend*. Seguidamente, se muestra la cabecera de un controlador:

```
using GenteFit.Models;
using GenteFit.Models.Repositories.Collections;
using GenteFit.Models.Repositories.Interfaces;
using GenteFit.Models.Usuarios;
using Microsoft.AspNetCore.Mvc;

namespace GenteFit.Controllers.ControllersMongoDB
{
    0 referencias
    public class ReservaController : Controller
    {
        // Instanciamos la interfaz del Modelo MongoDB
        private IReserva db = new ReservaCollection();

        // GET: ReservaController
        3 referencias
        public ActionResult Index()
        {
```

Traducción a objetos JSON

Como hemos comentado, MongoDB trabaja con objetos BSON, un tipo especial de objetos JSON. De este modo, tenemos que tener en cuenta que deberemos realizar la asignación necesaria dentro de nuestros métodos de inserción y modificación para poder crear los objetos que enviaremos a MongoDB. En este sentido, debemos ser conscientes de que será necesario parsear los tipos de datos que sean distintos a strings:

```
// Creamos un objeto para almacenar todos los datos que capturamos del formulario, debemos parsear
var centro = new Centro()
{
    Nombre = collection["Nombre"],
    Descripcion = collection["Descripcion"],
    Direccion = new Direccion(
        collection["Direccion.Domicilio"],
        collection["Direccion.Poblacion"],
        int.Parse(collection["Direccion.Cp"]),
        collection["Direccion.Pais"]
        ),
    Telefono = collection["Telefono"],
    Email = collection["Email"],
};
```

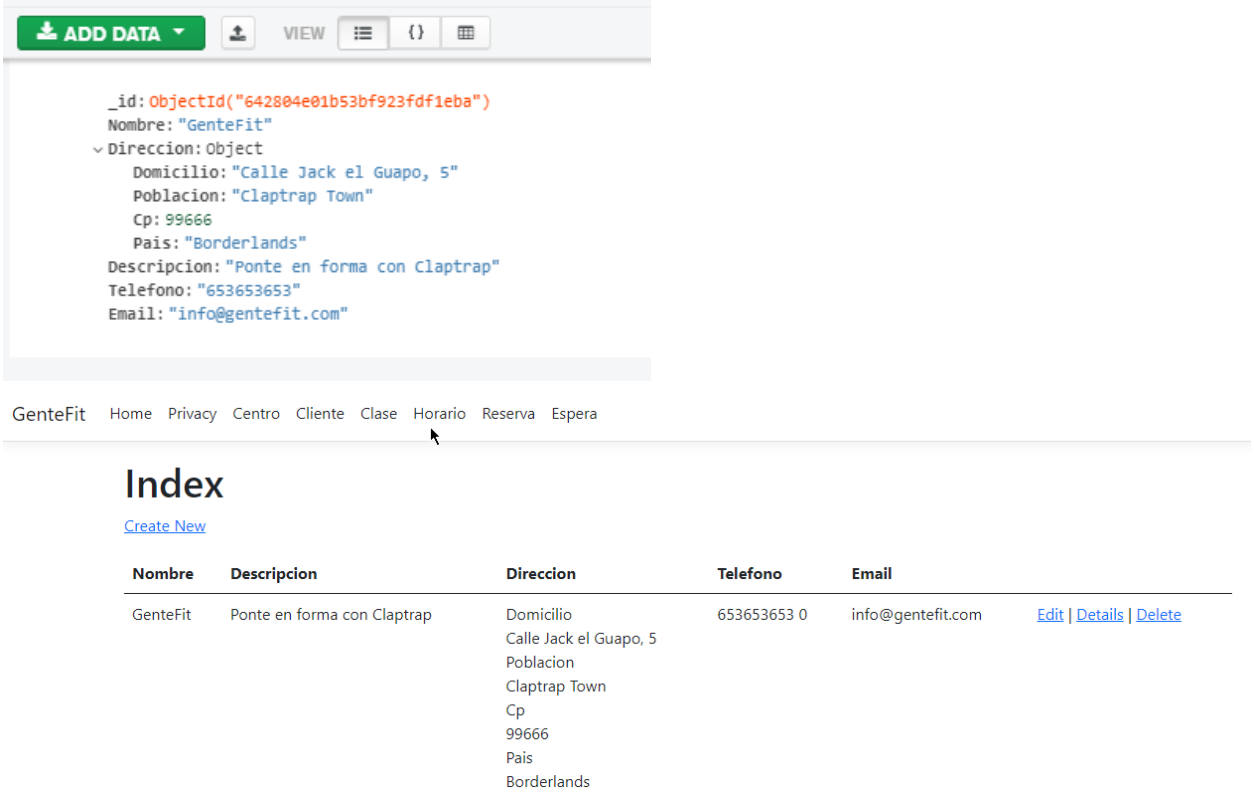
De igual modo, **MongoDB usa identificadores únicos para cada uno de los documentos que genera de forma automática asignándoles un código hash**. Ya que Mongo es una base de datos no-relacional, esto es muy importante porque va a ser el único modo que tendremos de identificar y manejar las asociaciones. Así, en nuestras clases para representar las entidades, debemos añadir una propiedad que maneje estos identificadores. Una vez más, la biblioteca MongoDB.Driver extiende una clase que se encargará de manejar todo este proceso. Así, nuestras clases tendrán un atributo Id del tipo ObjectId y precedidos por los especificadores necesarios para realizar la traducción de los tipos de datos:

```
[BsonId]
[BsonRepresentation(BsonType.ObjectId)]
8 referencias
public ObjectId Id { get; set; }
```

Comentarios finales

Como hemos señalado al principio, una de las mayores ventajas de MongoDB es que va a trabajar con objetos con una lógica de implementación muy similar a la de nuestro *backendk.* Así, veremos que no tendremos que abandonar nuestro paradigma de POO en ningún momento mientras estemos implementando nuestra app. Las colecciones de Mongo y nuestras clases para las entidades se van a comportar de la misma forma y vamos a gestionar el acceso a sus propiedades de igual manera, teniendo siempre una traducción directa de los objetos en ambas plataformas y no teniendo que batallar con las relaciones entre tablas. Evidentemente, esto que en nuestro caso es una ventaja, podría ser un problema en escenarios donde se tengan que usar BBDD con un peso muy grande de las relaciones entre tablas. Para nosotros no es importante porque la aproximación de Orientación a Objetos no es más que suficiente, manejaremos toda la lógica mediante las asociaciones entre clases.

En la siguiente imagen, Podemos ver como un documento de una colección se implementa como un objeto de POO:



Recursos

R. (2022, 28 noviembre). *Bibliotecas de Framework*. Microsoft Learn. https://learn.microsoft.com/es-es/dotnet/standard/framework-libraries

M. (2023, 21 febrero). *ADO.NET*. Microsoft Learn. https://learn.microsoft.com/es-es/dotnet/framework/data/adonet/

Construyendo un Web Api (I): Starter – Windows Platform. (2017, 11 enero). https://geeks.ms/windowsplatform/2017/01/11/construyendo-un-web-api-i/

The Coder Cave esp. (2020, 27 abril). *Crear CRUD Maestro/Detalle en MVC .NET Core y MongoDB*. YouTube. https://www.youtube.com/watch?v=YcNtrk-lEWU

Documentación Oficial de MongoDB, https://www.mongodb.com/docs/

QuickRef.ME. (s. f.). *C# Cheat Sheet & Quick Reference*. https://quickref.me/cs.html

D. (s. f.). *IComparable Interface (System)*. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/api/system.icomparable?view=net-8.0

Learn C# Properties: Getters and Setters at Intermediate C# Course. (s. f.). https://codeeasy.io/lesson/properties

Qué es un ORM (26 de febrero de 2018). Alarcón, José Manuel. https://www.campusmvp.es/recursos/post/que-es-un-orm.aspx