Laboratorio API REST

Laboratorio de prácticas tras la finalización de los módulos de .NET C#, API REST, Entity Framework, Test automatizados, Estructuración de soluciones y Autenticación/Autorización.

Descripción

El ejercicio consiste en la creación, desde cero, de un back-end en forma de API REST con las siguientes características:

- Framework MVC de ASP.NET Core con .NET 5
- Persistencia en una base de datos relacional SQL Server
- Entity Framework Core como ORM, siguiendo una estrategia *code-first* (primero el código, después la base de datos)
- EF CLI para la generación y aplicación de migraciones
- OpenAPI/Swagger para la descripción de la API REST
- Autenticación Básica con un middleware
- Tests unitarios con xUnit que abarquen también la integración con la base de datos

Pre-requisitos

- .NET 5 SDK
- Docker
- SQL Server como contenedor Docker (ver Anexo I)
- EF Core CLI (ver Anexo II)

Evaluación

La funcionalidad se divide en básica y opcional. Cada estudiante deberá crear un repositorio git en GitLab o GitHub e ir añadiendo las diferentes funcionalidades a su proyecto. Por ejemplo:

```
git add .
git commit -m "descripción_de_cambios"
git push
```

Se valorará la consecución de la funcionalidad básica y opcional, así como la calidad del código en cada uno de los commits del repositorio.

Funcionalidad

Funcionalidad Básica

1. Crea un repositorio git vacío en GitLab o GitHub, clónalo en tu máquina local y añade un .gitignore con `dotnet new gitignore`. Haz un primer push de tus cambios de la rama `main`.

2. Crea la estructura para una solución con 3 capas: WebApi, Application y Domain, donde WebApi es un proyecto de tipo Web y Application y Domain son proyectos de tipo class library. Por ejemplo, ejecuta los siguientes comandos en una terminal:

```
dotnet new classlib --name Lemoncode.Books.Domain --output
src/Lemoncode.Books.Domain
dotnet new classlib --name Lemoncode.Books.Application --output
src/Lemoncode.Books.Application
dotnet new web --name Lemoncode.Books.WebApi --output
src/Lemoncode.Books.WebApi

dotnet new xunit --name Lemoncode.Books.FunctionalTests --output
test/Lemoncode.Books.FunctionalTests

dotnet new sln

dotnet sln add src/Lemoncode.Books.Domain
dotnet sln add src/Lemoncode.Books.Application
dotnet sln add src/Lemoncode.Books.WebApi

dotnet sln add test/Lemoncode.Books.FunctionalTests
```

> Desde la consola se puede compilar toda la solución en cualquier momento con `dotnet build` y se pueden ejecutar los tests con `dotnet test`.

3. Abre la solución con Visual Studio y crea las siguientes dependencias entre proyectos:

```
Lemoncode.Books.Application referencia al proyecto Lemoncode.Books.Domain
Lemoncode.Books.WebApi referencia al proyecto Lemoncode.Books.Application
Lemoncode.Books.FunctionalTests referencia al proyecto
Lemoncode.Books.WebApi
```

> No olvides hacer un add/commit/push de tu código tan frecuentemente como consideres significativo, siempre y cuando no haya tests rotos o código que no compile.

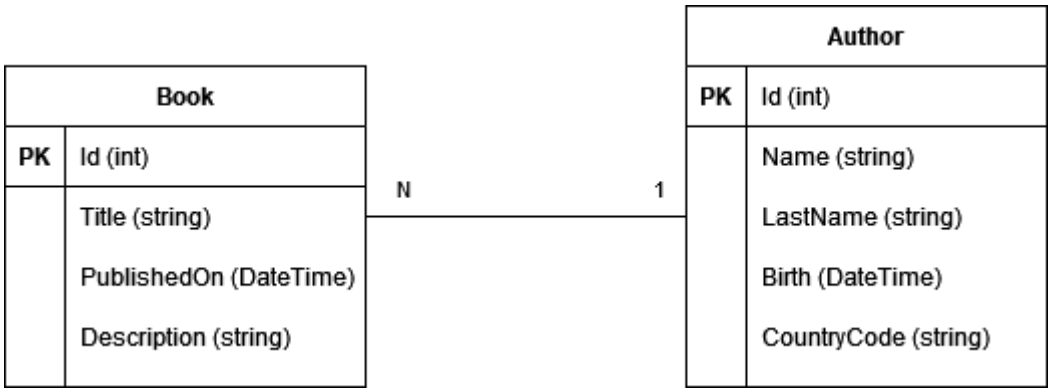
4. Vas a necesitar, entre otras, dependencias a paquetes nuget para utilizar Entity Framework Core. Los siguientes comandos muestran un ejemplo de cómo añadir dependencias nuget al proyecto deseado desde una terminal

```
dotnet add src/Lemoncode.Books.Application package
Microsoft.EntityFrameworkCore.SqlServer
dotnet add src/Lemoncode.Books.WebApi package
Microsoft.EntityFrameworkCore.Design
```

> Se recomienda añadir la dependencia al paquete nuget `Microsoft.EntityFrameworkCore.SqlServer` en la capa de aplicación, porque es ahí donde

> se creará el `DbContext`. De esta forma la capa de dominio se puede dejar totalmente agnóstica de paquetes externos y podrá ser utilizada para crear las entidades en ella. La dependencia al paquete nuget `Microsoft.EntityFrameworkCore.Design` se sitúa en el proyecto principal de web api, porque corresponde a la assembly ejecutable que el runtime de .NET necesita encontrar.

5. Ahora que ya tienes lo básico para comenzar a programar, y el conocimiento suficiente para añadir proyectos o dependencias si fuera necesario, crea las entidades y el `DbContext` (e.g: `BooksDbContext`) que permitan generar una base de datos en SQL Server con una estrategia *code first*. El modelo relacional se muestra en el siguiente diagrama.



   Se asume que un libro tiene un único autor y que un autor puede tener varios libros.

> Echa un vistazo al Anexo III en el que se describe cómo y dónde colocar la cadena de conexión a la base de datos, y al Anexo IV en el que se describe cómo se hacen las migraciones con EF Core CLI en una aplicación web.

6. Crear un endpoint para añadir Autores (i.e: `POST api/authors`) y que acepte payload en formato json para crear nuevos autores con nombre, apellido, fecha de nacimiento y nacionalidad como country code de dos caracteres *ISO 3166-1 alpha-2 code*

7. Crear un endpoint para añadir Libros (i.e: `POST api/books`) y que acepte payload en formato json para crear nuevos libros con título, fecha de publicación y descripción. Además debe aceptar el identificador del autor y asociarse con ese autor en concreto.

8. Crear un endpoint para modificar el título o la descripción de un libro (i.e: `PUT api/books/{bookId}`) dado su identificador único.

9. Crear un endpoint para consultar todos los libros de la base de datos (i.e: `GET api/books`) en el que se devuelve una respuesta en formato json que contiene una colección de objetos con la siguiente información

   - Id: el identificador único del libro
   - Title: el título del libro
   - Description: la descripción del libro
   - PublishedOn: la fecha en formato ISO_8601 UTC
   - Author: el nombre completo

10. Añade Swagger/OpenAPI (i.e: paquete nuget `Swashbuckle.AspNetCore`) para dar información sobre los diferentes endpoints

Funcionalidad Opcional

11. Añade un test funcional donde se valide que: Dada la creación por http de un autor, Cuando se lee de la base de datos el autor con el Id generado, Entonces devuelve el autor que se había creado
12. Añade un test funcional donde se valide que: Dada la creación de un autor que tiene dos libros, Cuando se lee a través del endpoint de http la lista de libros, Entonces debería haber dos, con los datos esperados tanto del autor como de los libros.
13. Añade autenticación básica con un middleware, configurable en el `appsettings.json` para proteger todos los endpoint y permitir que solamente las peticiones http que contengan una cabecera `Authorization: Basic <base_64_credentials>` puedan acceder
14. Modifica Swagger para conseguir que las peticiones http vayan siempre acompañadas de la cabecera `Authorization` con las credenciales esperadas.
15. Modifica los tests para que sigan funcionando con la autenticación básica.
16. Añade filtro como query parameter en `GET api/books?title=foo` que busque libros que contengan solamente lo indicado en el título.
17. Añade otro filtro como query parameter en `GET api/books?title=foo&author=bar` que busque libros que contengan solamente lo indicado en el título y además contenga el nombre o el apellido del autor indicado
18. Haz las mejoras y validaciones en el código que consideres oportunas

Anexo I - SQL Server con Docker

Crea un contenedor de SqlServer para tener un servidor de base de datos corriendo en local

```
docker run -e "ACCEPT_EULA=Y" -e "SA_PASSWORD=Lem0nCode!" -e "MSSQL_PID=Express" -
p 1433:1433 --name sqlserver -d mcr.microsoft.com/mssql/server:2017-latest-ubuntu
```

Asegúrate de que el contenedor está en ejecución con

```
docker ps
```

Si ya tenías ese contenedor (u otro), puedes simplemente arrancarlo

```
docker start sqlserver
```

> Opcionalmente instala descarga e instala Azure Data Studio para tu sistema operativo https://docs.microsoft.com/en-us/sql/azure-data-studio/download-azure-data-studio?view=sql-server-ver15

Anexo II - EF Core CLI

Instala el CLI de Entity Framework Core con

```
dotnet tool install --global dotnet-ef
```

o, si ya lo tienes instalado pero deseas tener la última versión, actualiza con

```
dotnet tool update --global dotnet-ef
```

Anexo III - Configuración de cadena de conexión en una aplicación web

EF Core CLI provee de una serie de herramientas para detectar cambios en el esquema relacional (i.e: en el `DbContext` y en las entidades que referencia) y permite sincronizarlos con una base de datos real a la que conectará si conoce el *connection string* (con el *hostname*, puerto, nombre de la base de datos y credenciales) a utilizar para acceder a ella.

Este *connection string* debería ser configurable, y no grabarse a fuego en el código ni requerir la recompilación cada vez que se cambie. Para ello, la mejor práctica es añadirlo en el `appsettings.json`. Por ejemplo:

```
"ConnectionStrings": {
    "BooksDatabase": "Server=localhost;Database=Books;user=sa;password=Lem0nCode!"
}
```

Estas *settings* se pueden leer en tiempo de ejecución gracias al servicio `IConfiguration` que se puede inyectar en el constructor de la clase `Startup`. El siguiente ejemplo muestra cómo se puede leer el connection string

```
public class Startup
{
    private readonly IConfiguration _configuration;

    public Startup(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    public void ConfigureServices(IServiceCollection services)
    {
        // aquí los registros de servicios en el contenedor de inyección de
dependencia. Por ejemplo para Entity Framework
        services.AddControllers();

        var booksConnectionString = _configuration.GetValue<string>
("ConnectionStrings:BooksDatabase");
        services.AddDbContext<BooksDbContext>(options =>
options.UseSqlServer(booksConnectionString));
    }
```

```
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        // aquí la pipeline de middleware a utilizar. Por ejemplo:
        app.UseDeveloperExceptionPage();
        app.UseRouting();
        app.UseAuthorization();
        app.UseEndpoints(endpoints => { endpoints.MapControllers(); });
    }
}
```

```
public class BooksDbContext
    : DbContext
{
    public BooksDbContext(DbContextOptions<BooksDbContext> options)
        : base(options)
    {
    }

    // Aquí los DbSet<YourEntity> que representan tablas

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
        // Aquí la configuración adicional para relaciones entre entidades
(solamente si hiciera falta)
    }
}
```

Anexo IV - Migraciones con EF Core CLI

Para crear una migración, o la inicial cuando se desea crear la base de datos, ejecuta desde el directorio raíz de la solución:

```
dotnet ef migrations add nombre_de_tu_migracion --project
src/Lemoncode.Books.Application --startup-project src/Lemoncode.Books.WebApi
```

Para aplicar las migraciones generadas que no hayan sido aplicadas en la base de datos aún, ejecuta:

```
dotnet ef database update --project src/Lemoncode.Books.Application --startup-
project src/Lemoncode.Books.WebApi
```

> Observa que en ambos casos se especifica unos parámetros. Con `--project` se le indica a EF Core CLI dónde encontrar el proyecto que tiene el `DbContext` (generalmente en la capa de aplicación, y las entidades en la capa de dominio), con `--startup-project` se le indica a EF Core CLI dónde encontrar

> el proyecto ejecutable de .NET que tiene una dependencia a `Microsoft.EntityFrameworkCore.Design`

Anexo V - Forzar el uso de una versión de .NET concreta

En noviembre de 2021, .NET 6 ha comenzado a estar disponible. Todos los ejemplos se pueden actualizar a .NET 6 sin problema, pero el ejercicio considera que tienes .NET 5 instalado (e.g: `5.0.403`).

Verifica la versión instalada con

```
dotnet --version
```

Si aparece que tienes la versión 6, puedes forzar a que se utilice la 5 (o cualquiera) creando el siguiente `global.json` en cualquier directorio para que todos los proyectos que cuelguen de él utilicen esa versión.

```
{
  "sdk": {
    "version": "5.0.403",
    "rollForward": "latestFeature"
  }
}
```

Recuerda que también debes utilizar Visual Studio 2019.