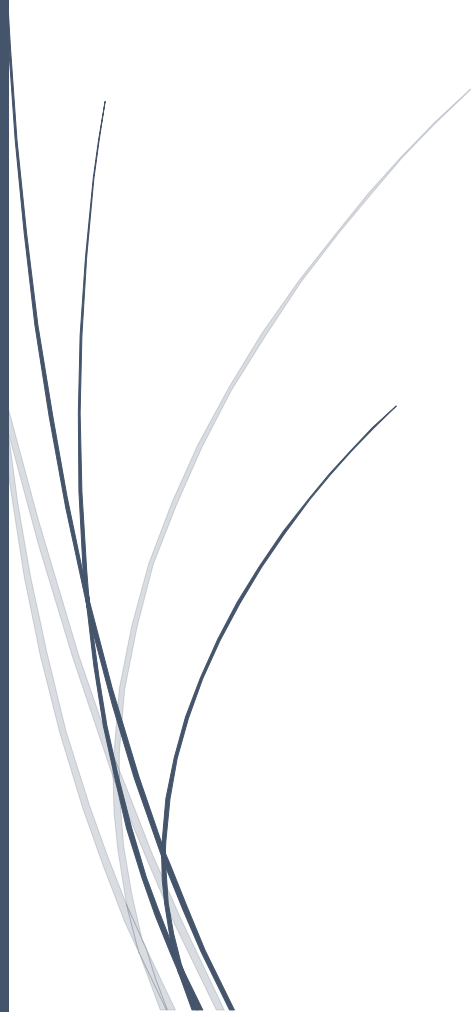
19-10-2023

Práctica 5

Diseño de la Base de Datos



Programación de Aplicaciones Móviles Nativas

GRUPO DE PRACTICAS 05

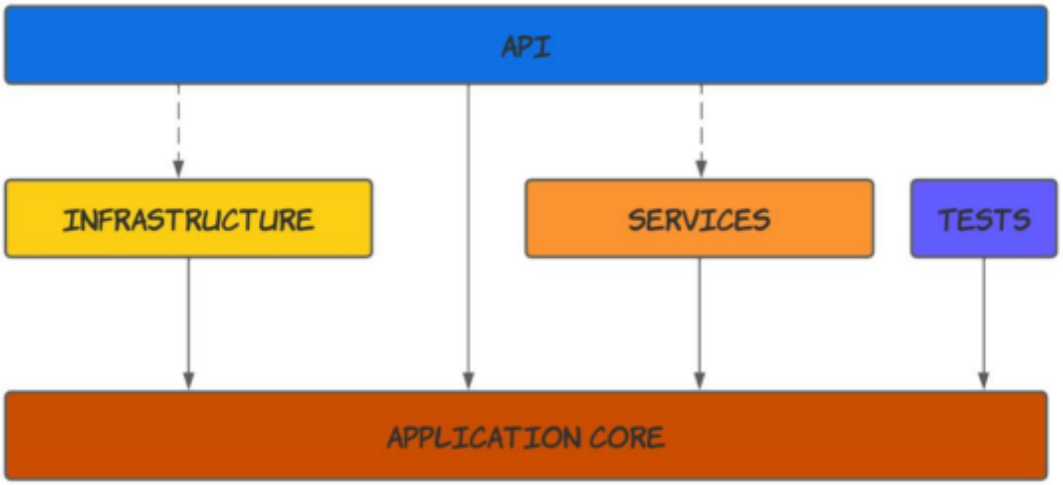
Contenido

1.- Revisa con tu equipo la arquitectura de tu proyecto final, identifica y resalta los modelos y componentes relacionados con la capa de persistencia.

En primera instancia habíamos definido la arquitectura a aplicar, que no es otra que 'Clean Architecture' una vez aplicada a nuestro proyecto, que recordemos que tenemos por un lado el **Frontend/Cliente** una app Android y como **Backend** una API desarrollada con *ASP.NET* en esta última es donde se aplica la arquitectura dado que contiene toda la lógica.

Una vez aplicada la arquitectura el backend nos queda de esta manera:

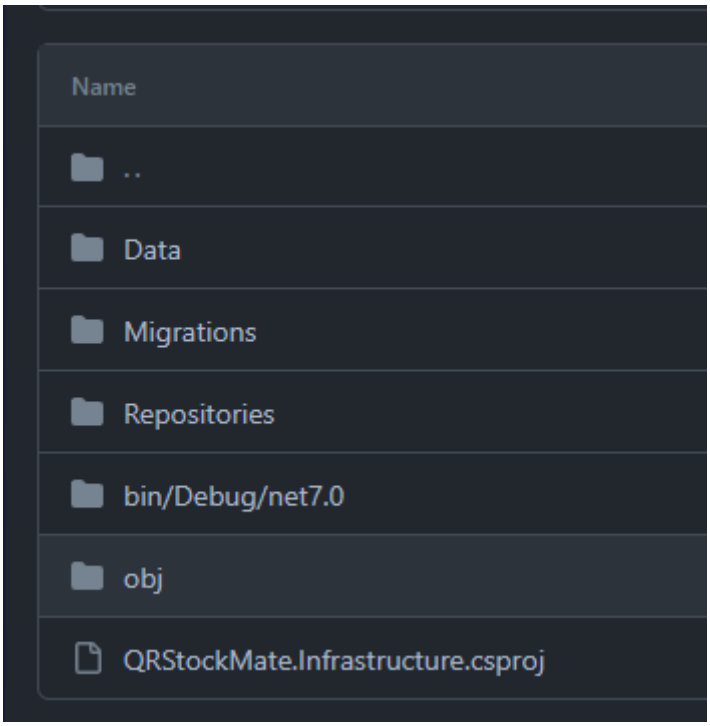
Aquí vamos a resaltar las relaciones, para destacar la dependencia entre capas y la separación de responsabilidades, en el diagrama original teníamos lo siguiente:



En nuestro caso la asignación seria la siguiente:

API	*-> QRStockMate*
INFRASTRUCTURE	*-> QRStockMate.Infrastructure*
SERVICES	*-> QRStockMate.Service*
APLICATION CORE	*-> QRStockMate.AplicationCore*

En este caso nos vamos a centrar en la capa de persistencia que es la referente a la capa de *'INFRASTRUCTURE'* su contenido es el siguiente:

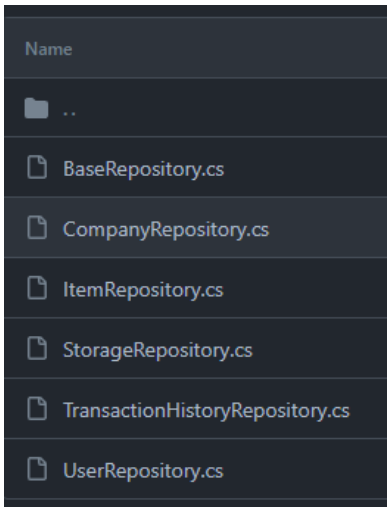


1. Data, contiene el archivo que hace referencia a la base de datos, a las tablas y a los tipos que la representan

```
namespace QRStockMate.Infrastructure.Data
{
    public class ApplicationDbContext:DbContext
    {

        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options):base(options)
        {
        }

        public DbSet<User> Users{ get; set; } = null!;
        public DbSet<Company> Companies { get; set; } = null!;

        public DbSet<Item> Items { get; set; } = null!;
        public DbSet<TransactionHistory> TransactionsHistory { get; set; } = null!;
    }
}
```

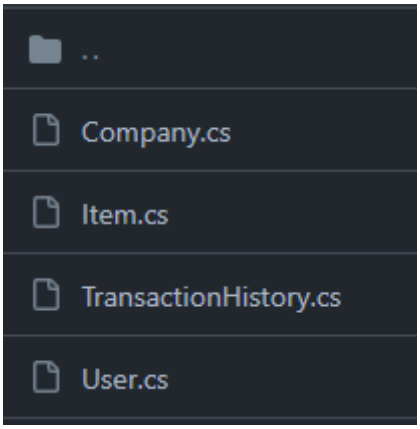
2. Migrations, contiene todas las migraciones que se han realizado en la base de datos para cualquier cambio en el futuro sobre las entidades implicadas en ella, de manera que esta sea totalmente escalable.

3. Repositories, contiene los componentes que interactúan con las distintas operaciones que se aplican sobre la base de datos para las distintas entidades existentes.

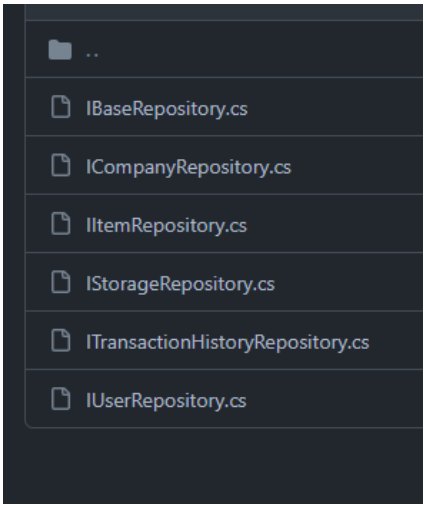


Los 2 últimos elementos se encuentran en la capa de '*AplicationCore'*, y este es utilizado por la capa de *'Infrastructure'*, debido a la dependencia observada en el diagrama:

4. Entities, este contiene todas las entidades necesarias que conforman la lógica de negocio y que por tanto es utilizada por la capa de *'Infrastructure'*



5. Interfaces, este contiene todas las operaciones exclusivas y en común de las distintas entidades sobre la propia base de datos.

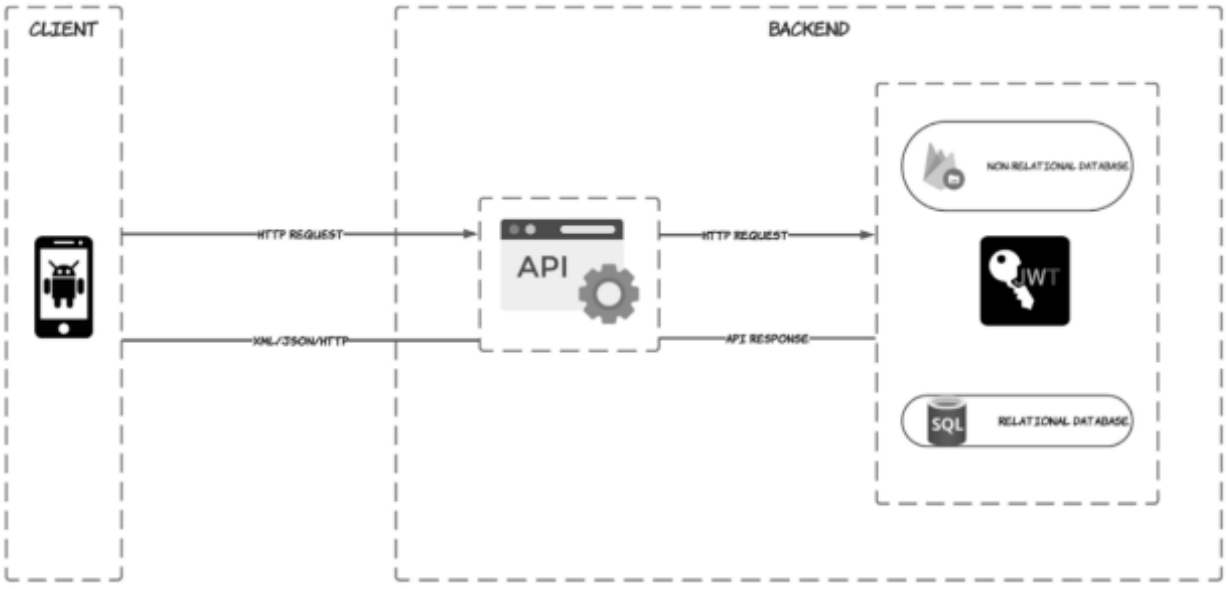


2.- Selecciona una tecnología para la base de datos de tu aplicación (Room, Firebase, SQLite, Realm, DataStore ...etc). Justifica brevemente tu elección, considerando las necesidades de tu aplicación.

Como se ha comentado enteriormente las bases de datos a utilizar en el proyecto son

- **SqlServer**, ofrece características como transacciones ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad), soporte para consultas SQL, seguridad avanzada y escalabilidad. Es ideal para aplicaciones empresariales, aplicaciones de gestión de datos y sistemas que requieran alta confiabilidad y rendimiento, justo lo que requiere nuestro proyecto.
- **FirebaseStorage**, base de datos no sql cuyo uso principal es guardar archivos y fotos además contiene CDN integrado para la entrega rápida de contenido y la posibilidad de acceder a los archivos desde cualquier lugar
- **Sqlite,** para la capa de fronted, es decir la aplicación movil, si se requiere para guardar alguna informacion de preferencias de usuarios etc es posible que se implemente, **pero no esta en nuestros planes su uso** ya que no se ve necesario hasta el momento.

Para un mejor entendimeinto de como se integraria hago referencia a la imagen de nuestra arquitectura



3.- Define cómo se integrará la lógica que gestionará la capa de persistencia en tu proyecto. Puedes complementar el diagrama empleado en la actividad sobre la arquitectura de tu aplicación. Esto puede incluir entidades, atributos, relaciones, DAOs, repositorios. Importante: bastaría con un sólo módulo de la aplicación

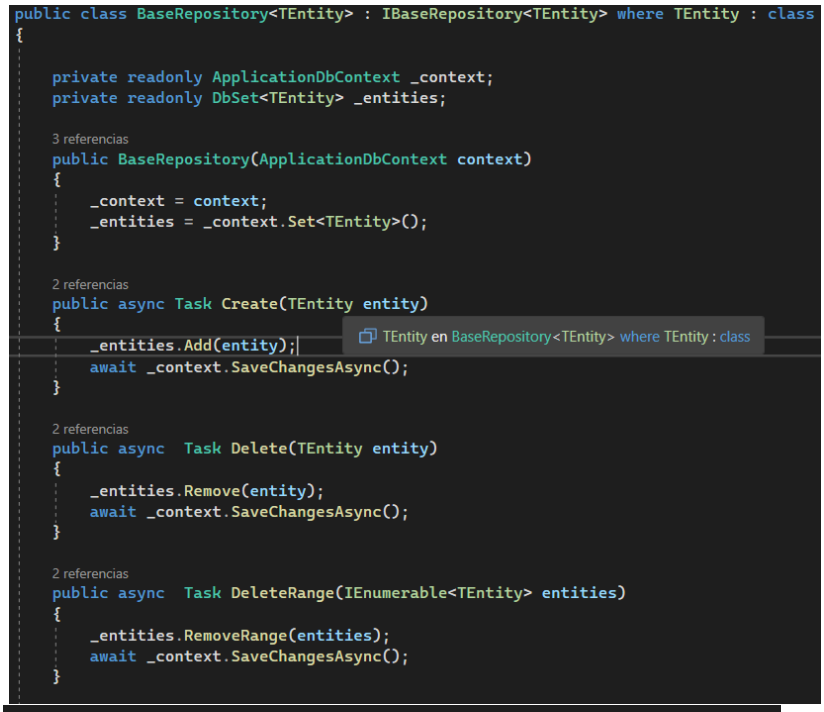
En nuestro proyecto, la gestión de la capa de persistencia se llevará a cabo de manera eficiente y estructurada. Para lograr esto, hemos diseñado un proceso claro y organizado que asegura la correcta manipulación de los datos y su almacenamiento en la base de datos. Este proceso se integra de la siguiente manera:

1. **Modelo a Entidad:** Cuando se recibe una petición que involucra datos, el primer paso es transformar los datos del modelo de la aplicación en entidades. Las entidades son representaciones de los objetos que serán almacenados en la base de datos. Esta transformación garantiza que los datos estén en el formato adecuado y cumplan con las restricciones definidas para la persistencia. En la siguiente figura podemos ver un ejemplo de la función get del controlador de usuario.

```
[HttpGet]
0 referencias
public async Task<ActionResult<IEnumerable<UserModel>>> Get()
{
    try
    {
        var users = await _userService.GetAll();

        if (users is null) return NotFound();//404

        return Ok(_mapper.Map<IEnumerable<User>, IEnumerable<UserModel>>(users)); //200
    }
    catch (Exception ex)
    {

        return BadRequest(ex.Message);//400
    }
}
```

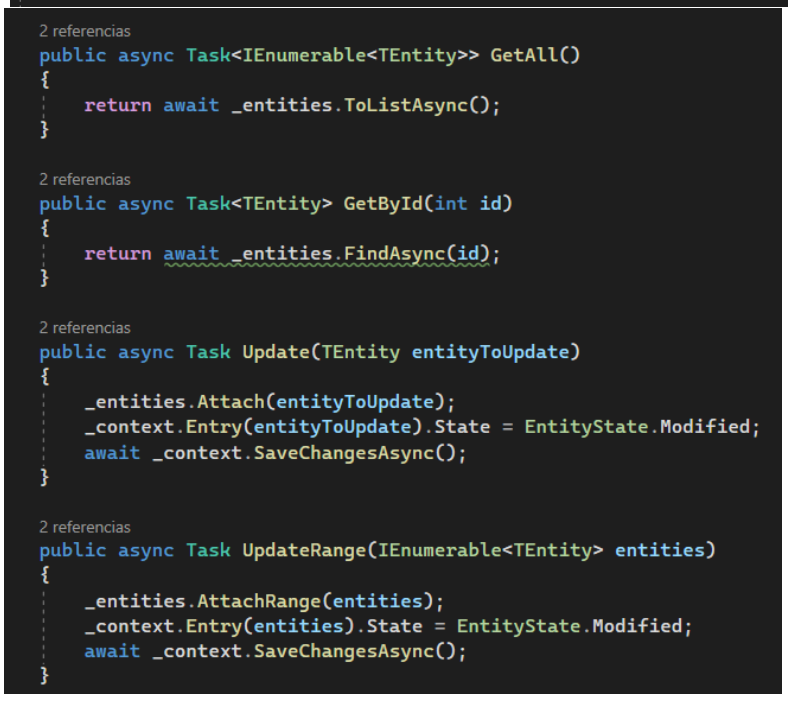
2. **Llamada a Servicios:** Posteriormente se llama al servicio correspondiente. Estos servicios ayudan a garantizar que los datos sean coherentes y cumplan con reglas de negocio específicas antes de continuar con la persistencia. En la figura anterior el controlador llama a la función GetAll() del servicio de la entidad usuario.
3. **Acceso a Repositorio:** Una vez que los datos han pasado las validaciones y conversiones necesarias, se procede a acceder al repositorio de datos. El repositorio es la capa que interactúa directamente con la base de datos y se encarga de realizar operaciones de lectura, escritura, actualización y eliminación.

Nuestro sistema se basa en una clase abstracta llamada '*BaseRepository*' que actúa como una interfaz común para interactuar con la base de datos. Esta clase contiene métodos genéricos para realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) en la base de datos. Al utilizar esta clase como base, logramos una gestión uniforme y eficiente de las operaciones de persistencia en toda la aplicación, ya que todos los repositorios la implementan. Las siguientes figuras son capturas del 'BaseRepository' que implementa todas las funciones básicas (CRUD) que sirve para todos los repositorios específicos.



```
public class BaseRepository<TEntity> : IBaseRepository<TEntity> where TEntity : class
{

    private readonly ApplicationDbContext _context;
    private readonly DbSet<TEntity> _entities;

    3 referencias
    public BaseRepository(ApplicationDbContext context)
    {
        _context = context;
        _entities = _context.Set<TEntity>();
    }

    2 referencias
    public async Task Create(TEntity entity)
    {
        _entities.Add(entity);
        await _context.SaveChangesAsync();
    }

    2 referencias
    public async  Task Delete(TEntity entity)
    {
        _entities.Remove(entity);
        await _context.SaveChangesAsync();
    }

    2 referencias
    public async  Task DeleteRange(IEnumerable<TEntity> entities)
    {
        _entities.RemoveRange(entities);
        await _context.SaveChangesAsync();
    }
```

TEntity en BaseRepository<TEntity> where TEntity : class



```
    2 referencias
    public async Task<IEnumerable<TEntity>> GetAll()
    {
        return await _entities.ToListAsync();
    }

    2 referencias
    public async Task<TEntity> GetById(int id)
    {
        return await _entities.FindAsync(id);
    }

    2 referencias
    public async Task Update(TEntity entityToUpdate)
    {
        _entities.Attach(entityToUpdate);
        _context.Entry(entityToUpdate).State = EntityState.Modified;
        await _context.SaveChangesAsync();
    }

    2 referencias
    public async Task UpdateRange(IEnumerable<TEntity> entities)
    {
        _entities.AttachRange(entities);
        _context.Entry(entities).State = EntityState.Modified;
        await _context.SaveChangesAsync();
    }
```

4.- Reflexiona sobre cambios futuros:

- ¿Qué pasa si en un futuro se quisiera cambiar el motor de base de datos?

- Al tener aplicada la **Arquitectura Limpia** solamente tendriamos que `ojear` la capa de Infrastructure y digo ojear porque al trabajar con funciones abstractas sobre entidades no definidas solo habria que cambiar alguna que otra función de busquedad que sí seria necesario dependiendo del motor de busquedad.

- ¿Qué partes de tu aplicación tendrías que modificar?

- Como comenté anteriormente tenemos el proyecto estructurado de tal forma que las dependencias entre capas son minimas, en este caso habria que modificar la capa de Infrastructure y ya está, y realmente no haria falta porque como dije *'al trabajar con funciones abstractas sobre entidades no definidas solo habria que cambiar alguna que otra función de busquedad que sí seria necesario dependiendo del motor de busquedad'*

- ¿Qué dificultades anticipas?

- Las dificultades pueden surgir si el nuevo motor de base de datos tiene diferencias significativas en la sintaxis SQL o en las características admitidas.

- ¿Cómo podrías diseñar tu aplicación para minimizar el impacto de tal cambio?

- **Utilización de Interfaces**: Gracias a las interfaces, las clases de los repositorios y servicios están desacopladas de la implementación concreta del motor de base de datos. Esto facilita enormemente la creación de nuevas implementaciones adaptadas al nuevo motor.
- **Abstracción de la Base de Datos**: Con la capa de abstracción que ya hemos incorporado, podemos ocultar los detalles específicos del motor de base de datos. Esto simplifica la transición y minimiza el impacto en las capas superiores de la aplicación.
- **Documentación**: Con la arquitectura limpia, documentamos nuestras interfaces y componentes de manera efectiva usando Swagger por defecto. Esto facilita la comunicación con el equipo y garantiza que todos estén informados sobre las modificaciones necesarias.