Technical Test Crud API

The purpose of this document is to outline the design strategy for implementing CRUD (Create, Read, Update, Delete) operations in an ASP.NET Core API for a given schema. Additionally, it describes the integration of a retry and backoff mechanism for database write operations to enhance the robustness of the system.

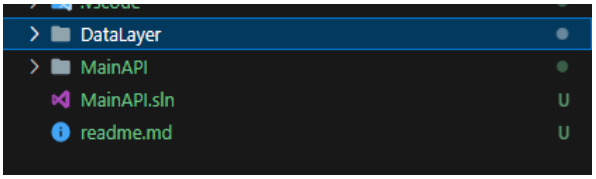
Database schema:

The provided schema consists of entities representing an individual with attributes such as addresses, names, and dates. These entities have relationships with each other, forming a relational database structure.

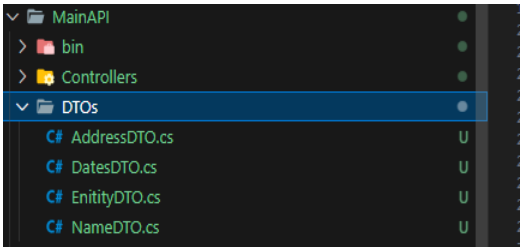
I have used Fluent API approach to design the database schema using code first approach using Entity Framework Core.

```
        0 references
        public DbSet<Entity> Entities { get; set; }
        0 references
        public DbSet<Address> Addresses { get; set; }
        0 references
        public DbSet<Date> Dates { get; set; }
        0 references
        public DbSet<Name> Names { get; set; }

        0 references
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Entity>()
                .HasMany(e => e.Addresses)
                .WithOne(a => a.Entity)
                .HasForeignKey(a => a.EntityId)
                .IsRequired();

            modelBuilder.Entity<Entity>()
                .HasMany(e => e.Dates)
                .WithOne(d=>d.Entity)
                .HasForeignKey(d => d.EntityId)
                .IsRequired();

            modelBuilder.Entity<Entity>()
                .HasMany(e => e.Names)
                .WithOne(n=>n.Entity)
                .HasForeignKey(n => n.EntityId)
                .IsRequired();
        }
    }
}
```

I used a separate layer for database interaction to abstract the complex logic of reading and writing data to database.



DTO (data transfer objects):
In the data layer, entities are kept in separate directory. I also used DTOs for keeping my entities separate from actual entity so that I can prevent the actual entity exposing to outside of this layer.



Then I implemented the complete CRUD in the repository class, and called these methods in the api endpoints in the controller.

1. Create:
This method takes entity model as parameter and add to database (create new entry).

```
2 references
public async Task<bool> CreateEntity(Entity entity)
{

    _dbcontext.Entities.Add(entity);

    var res = await SaveChangesWithRetry();
    return res;
}
```

2. Read all data (without any filters):
This Method lists all the data from the database and also loads the related entities such as name, address and date.

```
2 references
public async Task<IEnumerable<Entity>> GetEntities()
{
    return await _dbcontext.Entities
                .Include(e => e.Addresses)
                .Include(e => e.Names)
                .Include(e => e.Dates)
                .ToListAsync();
}
```

3. Read single entity by id:
This method takes id as parameter, find from the database and return if found.

```
3 references
public async Task<Entity> GetEntityById(string id)
{
    var res = await _dbcontext.Entities.Where(e => e.Id == id)
                        .Include(e => e.Addresses)
                        .Include(e => e.Names)
                        .Include(e => e.Dates)
                        .FirstOrDefaultAsync();

    return res;
}
```

4. Get all data with searching, filtering and page No. with page size:

This method takes many optional parameter such as searchQuery, and fiter keys like gender, startDate, endDate, etc., and also pageNo, pageSize, and orderBy for sorting.

In this method I used IQueryable object for filtering one by one on each key only if they are passed as parameter and not null.

```
var query = from e in _dbcontext.Entities select e;

if (!string.IsNullOrEmpty(searchQuery))
{
    query = query.Where(e =>
        e.Addresses.Any(a => a.Country.Contains(searchQuery) || a.AddressLine.Contains(searchQuery)) ||
        e.Names.Any(n => n.FirstName.Contains(searchQuery) || n.MiddleName.Contains(searchQuery) || n.Surname.Contains(searchQuery))
    );
}

if (!string.IsNullOrEmpty(gender))
{
    query = query.Where(e => e.Gender == gender);
}

if (startDate != null && endDate != null)
{
    query = query.Where(e => e.Dates.Any(d => d._Date >= startDate && d._Date <= endDate));
}

if (countries != null && countries.Any())
{
    query = query.Where(e => e.Addresses.Any(a => countries.Contains(a.Country)));
}
```

Then after filtering and searching I applied ordering based on orderBy text passed in parameter. Then paging is used to skip data and take only current page data of page size length.

```
if (orderBy != null)
{
    switch (orderBy.ToLower())
    {
        case "name":
            query = query.OrderBy(e => e.Names.FirstOrDefault().FirstName);
            break;
        case "date":
            query = query.OrderBy(e => e.Dates.FirstOrDefault()._Date);
            break;
        case "country":
            query = query.OrderBy(e => e.Addresses.FirstOrDefault().Country);
            break;
    }
}
```

At last a single query will be sent to database to fetch the filtered data finally from the database making the process faster and not making request for separate filters.
(That is why I called the ToList in the last after applying al filters on the IQueryable instance)

```
    pageSize = pageSize > 20 ? 5 : pageSize;

    if (pageNo > 0)
    {
        query = query.Skip((pageNo - 1) * pageSize).Take(pageSize);
    }


    var data = query.Include(e => e.Addresses)
                    .Include(e => e.Names)
                    .Include(e => e.Dates);

    return data;
}
```

5. Update Entity:
This method updates the entity if it exits (check by Id).

```
2 references
public async Task<bool> UpdateEntity(Entity entity)
{
    bool ifExists = _dbcontext.Entities.Any(e => e.Id == entity.Id);

    if (!ifExists)
        return false;

    _dbcontext.Entities.Update(entity);

    var res = await SaveChangesWithRetry();
    return res;
}
```

6. Delete Entity:
This method deletes an entity from the database. (A check for the presence of the record to delete is performed in the controller and then deleted. If not found returned 404 error)

```
2 references
public async Task<bool> DeleteEntity(Entity entity)
{
    _dbcontext.Entities.Remove(entity);
    var res = await SaveChangesWithRetry();
    return res;
}
```

7. Saving Data to database with retry and backoff mechanism:

This method is called in all the write cases of database like adding, updating, and deleting. This method is implemented with retry and backoff mechanism for consistency in case of failures.

I have used exponential delay, in seconds with maximum of 3 attempts. In the first attempt it try to write the database, and if failed (any exception occurred, delay is increased exponentially for next attempt. This approach is used to protect continuous bombarding of requests to database.

```
public async Task<bool> SaveChangesWithRetry()
{
    int max_retry = 3;
    int current_attempt = 1;

    TimeSpan delay = TimeSpan.FromSeconds(1.5);

    while (current_attempt <= max_retry)
    {
        try
        {
            await _dbcontext.SaveChangesAsync();
            _logger.LogInformation($"Database Write Successfull at attempt = {current_attempt}");

            return true;

        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error occurred during database write operation. Retrying...");


            _logger.LogInformation($"Retry attempt {current_attempt} in {delay} seconds.");
```

Finally I called all these methods for different endpoints in the controller.I used the repository as dependency injection in the controller. This makes it more easier to

maintain and modify because everything related to database is dealt at one place (repository class).