# A Simple Trader Service

## List of Technologies Used

* .NET 5
* ASP.NET MVC
* GraphQL, including solution to N + 1 problem
* Dependency injection
* Repo pattern
* Synchronization for async/await methods
* Transport Layer Security (TLS)
* ORM *EntityFrameworkCore*
* OpenApi (a.k.a. Swagger)
* In-memory service for integration tests

## Intro

The service provides simple handling (base CRUD operations) of traders list. It is based on [GraphQL](https://en.wikipedia.org/wiki/GraphQL) technology. GraphQL allows to retrieve and submit stored data in a very ordered and yet flexible manner. It provides schema acting as a contract between client and server. The schema also defines retrieval procedure in the server.

## GraphQL

GraphQL defines contract between client and server for data retrieval (query) and update (mutation). Both query and mutation constitute JSON-like structure. Retrieved data are formed into much the same structure as request and return to client. Due to hierarchical form of GraphQL query process of data retrieving is a sequence of calls to handlers of nested fields.

## Query N + 1 Problem

Usage of GraphQL brings about so called query [N + 1 problem](https://stackoverflow.com/questions/97197/what-is-the-n1-selects-problem-in-orm-object-relational-mapping#:~:text=The%20N%2B1%20query%20problem,the%20larger%20the%20performance%20impact.). Efficient GraphQL application has to provide reasonable solution to this issue. In this solution I implemented my original code solving N + 1 problem for GraphQL. In few words it may be formulated as follows. In "naive" solution handler of every field calls database to retrieve data. In optimized solution the first call of field handler on each level retrieves from database data for all fields on this level and stores them in a cache attached to GraphQL context object available to all field handlers. Subsequent calls of the given level field handler obtain data from the cache and not from database. The difference is illustrated by Figs. 1 and 2.

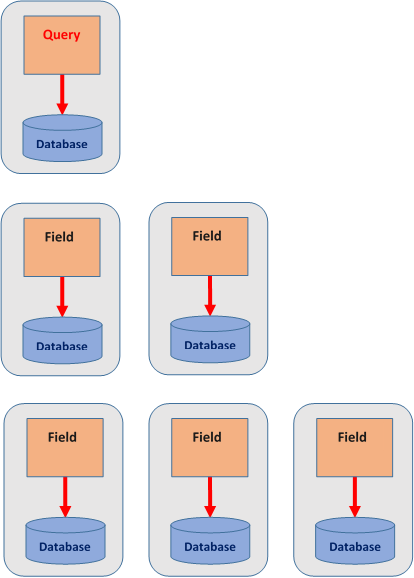


Fig. 1. Non-Optimized GraphQL

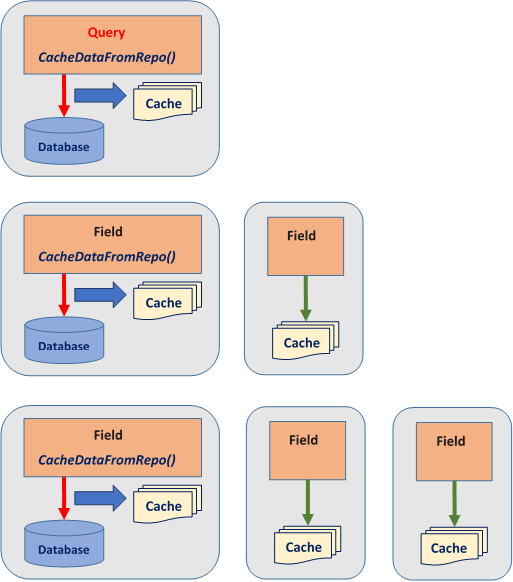


Fig. 2. Optimized GraphQL

Similar approach is discussed in details and was implemented with Node.js in [my article](https://www.codeproject.com/Articles/5289928/Node-js-Service-with-NestJS-and-GraphQL).

Cache may be organized in different ways. It may be local in-memory cache, as it is implemented in this solution. And in case of large amount of data it may be distributed cache based on, say, *Redis* or other similar products. That is why cache is used with interface *IGqlCache.* And its simple in-memory implementation *GqlCache : IGqlCache* is used in this solution.

## Components and Structure

|  |  |  |
| --- | --- | --- |
| **Component** | **Project Type** | **Description** |
| TraderService | Service (Console application) | Provides two controllers. *GqlController* processes all GraphQL requests, whereas *TraderController* processes parameterless GET request responding with some predefined “about” text, and another GET request with Trader *id* as a parameter. This request is internally processed as an ordinary GraphQL request with hardcoded query. It acts as a “shortcut” to often used GraphQL query. *TraderController* serves just illustrative purpose. The service referred to several general purposes libraries (DLLs). They are located in directory *.\Libs*. |
| RepoInterfacesLib | DLL | Provides interfaces to deal with data repository. |
| RepoLib | DLL | Implements *IRepo<T>* interface from *RepoInterfacesLib* for *EntityFramework*Core. It equips data saving procedure with transaction. |
| GraphQlHelperLib | DLL | Contains general GraphQL related software including code for data caching discussed above. |
| AsyncLockLib | DLL | Provides locking mechanism for async/await methods, particularly applied in usage of caching. |
| TraderModelLib | DLL | The project provides code specific for the given domain problem. Located in directory *.\Model .* |
| TraderServiceTest | Test Project (Console application) | Project provids integration tests for *TraderService*. These tests are based on the concept of in-memory service. Such an approach allows developer effortlessly test actual service code. Located in directory *.\Test* . |

## Features

* Very flexible formation of queries and insertion / updates due to usage of GraphQL.
* “Soft” deletion of traders – by toggling boolean *isDeleted* field in Traders database table.
* “Shortcut” to often used GraphQL queries with ordinary GET request in usual (non-GraphQL) controller.
* Handy *Playground* Web UI to write queries and mutations out-of-the-box with no code required.
* Usage of OpenApi (Swagger).
* Flexible configurable logging (currently configured for some minimum output to console only).
* Integration tests using in-memory service.

## Further Development

* *Database handling enhancement*. For easy start, Code First approach to database was adopted. Records IDs are generated from code which is problematic in real world application, e.g. due to possible insertion to database by several instances of a service or by several services. Some sequence solution should be implemented.
* Sorting and pagination should be more general. One of possible approaches is described in [this my article](https://www.codeproject.com/Articles/1234306/Sorted-Page-from-Distributed-Unsorted-Data-Sets).
* Encrypted password.
* *Authentication* (I have already implemented JSON Web Token (JWT) authentication for .NET 5 services but did not include it to this solution yet).
* More tests, e. g. for unusual business cases (“border cases”) and database operations.
* Deployment with *Docker*.