TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Ranno Rajaste 185564IADB

JIM WENDLER'S 5/3/1 PLANNER

Scope of work in Distributed Systems project

Supervisor: Andres Käver

Tallinn 2018

Table of contents

Tallinn 2018

1 Project description

The core idea for this project is to build a flexible workout routine planner for the popular powerlifting routine „5/3/1” by Jim Wendler. „5/3/1” is one of the most popular intermediate level non-linear cyclic barbell freeweight workout routines in the world, consisting of “workout cycles” with varying ranges of repetitions and weights depending on the week of the cycle. Due to the inherent complexity of programming for this routine it would be helpful to be able to plan out and program workouts with the assist of a workout planner application. At the moment no such popularly used accessible planner exists and most users rely on complicated and difficult to use spreadsheets to map out their cycles.

This project aims to provide the user with an easy to access web application for personal “5/3/1” programming as well as result tracking for both nutrition intakes and overall body composition.

2 Scope of work

2.1 Overview

The scope of work for this project is to build a distributed web application that allows the user to:

- Create a planned personal workout program based on Jim Wendler's "5/3/1" routine
- Calculate estimated one-repetition maxes for core barbell lifts based on several popular methods for one-repetition max calculation
- Track daily caloric and macro nutrient intake
- Track changes in body composition as well as generate expected results for composition changes over various periods
- Create and publish various workout routine templates that as an administrator
- Manage, create and lock/unlock user accounts as an administrator

2.2 Project structure

2.2.1 Frontend user interface

This project has two frontend user interfaces: server-side generated web pages for administrator views using *ASP.NET MVC* technologies and a *Javascript* single page application built in *Vue.js* framework for regular users.

Both user interfaces provide granular resource protection and security for users as well as internationalization and localization support both in user interface level and database content level.

2.2.2 Backend API

Backend for this project is developed using distributed system architecture, where information is kept in a remote *SQL Server* database and accessed and processed by a *.NET Core backend* application in a web server, which provides information exchange to separately hosted *Javascript* front-end via REST endpoints. REST endpoints are secured by *JSON web token* authenticated access.

3 Analysis

3.1 Database design

3.1.1 General design and technologies

Since one of the key requirements for this application is to be scalable and future proof, it is necessary to choose an appropriately complimenting database software and provider. In this project the choice is Microsoft's *SQL Server*, since it provides complex support for database operations, such as procedures for database level soft delete implementation and data backups, and has first level support for *Entity framework core* which makes interfacing the rest of the .NET webapplication with the database easier and more managable.

Database design for this project is in standard *Entity relationship diagram* form. The generated *ERD* is, however, not used for database initialization, the latter is done in *code first* design strategy, where *Entity Framework* is set up and configured with domain objects and a database migrations are generated from those, which are applied to the database. To keep the database as small as possible without storing excess reduntant information, the database entities are normalized as to the third normal form at the cost of somewhat more complex queries, especially on the administrator side, where each part of a *Workout routine* has to be checked for user ownership before being queried out.

3.1.2 Data seeding

Since several of the project database tables contain information that needs to be preloaded, optional data seeding on application startup is implemented using *JSON* to *C#* object converter provided by *NewtonSoft*. Seeding data is entered into a designated directory as *JSON* files, which are parsed into domain objects and directly saved into database using the *AppDbContext* on application startup. Data seeding and other database preoperations can be configured in the *appsettings.json* file

3.1.3 Soft delete implementation

In order to enable database auditing and reversable changes to data it is necessary to implement some form of *logical delete*, also known as *soft delete* in the project's database, meaning no records are actually permanently deleted from the database, instead they are marked as '*deleted*' and simply filtered out when querying for data.

To achieve a complete *time shift* effect i.e to be able to change the current time and to see the database at the exact state on that given time I have decided to implement *soft delete* using metadata fields '*created_at*' and '*deleted_at*' on all entities.

When a record is created, its *'created_at'* field is given the value of current *datetime* and its *'deleted_at'* is set to maximum possible *datetime* value. The latter is done in order to give an arbitrary date of deletion to all currently active undeleted records. The reason for this is to avoid potential issues that would arise from comparing dates in database with *null*. When a record is later deleted, its *'deleted_at'* field is set to current *datetime.* This allows queries to return records that were created before and deleted after the current date and time, unfortunately also meaning that every single query has to include comparison of *'deleted_at'* and *'created_at'* against the current *datetime*.

By default my implementation of *soft delete* does not cascade, this is done because of the philosophy that a closed entity might still have active related entites. For example if an app user has decided to close their account it does not mean that every single post they made has to be deleted from the website also.

In addition *soft update* is also implemented. For dependent entities when a record is updated, an exact copy of the record is created first and the copy's *'deleted_at'* field is changed to current *datetime.* Then the original records '*created_at'* date is changed to

current *datetime* along with any updates required to any other information fields. The primary key for both records is one field and is unique. For dependent entities updating is, in essence, simply creating a copy of the original record, closing it, and then applying changes to the original record.

Soft updating principal entities is more complex due to foreign key integrity constraint. *Soft updating* for principal entities is done by making the primary key of principal entities composite from fields ’*<Entity name>Id*’ and ’*deleted_at*’ from said principal entity. This primary key is used by all dependent entities as foreign key. A record is then updated exactly the same as in dependent entities. The result of this is that all dependent records now point to the most recent record. A significant downside of this approach is that even though regular present-day queries work as usual with comparison of both foreign key fields (*<entity_name>id* and *deleted_at*) guaranteeing foreign key integrity, audition queries from the past can only compare one field of the foreign key (*<entity_name>id* field). This theoretically breaks foreign key integrity, but since all records were added into the database under the constraint, all historical records technically still have this integrity, and if queries are filtered properly by comparing current date to *deleted_at* and *created_at* then no data anomalies should arise.

The described approach works well for *one-to-many* type relations between entities, however, for ‘1:0-1’ type relations, it is necessary to force uniqueness on *‘deleted_at’* field instead of the usual foreign key field. This means that there can be several records with the same foreign key allowing for record updating, but only one of them can be active, since no two records can exist with the same maximum possible value for *‘deleted_at’*, which marks records as ‘current’. This means that, for example, a user can have several ‘deleted’ profile pictures, but only one active one. This can fail, however, in a situation when a new record is entered into the dependent table with different maximum value for ‘*deleted_at*’. This can happen, for example, when changing database providers or when someone manually enters a date for ‘*deleted_at*’. The result is two different active profile pictures showing up in a query instead of one. For this reason until another solution is implemented that fixes this issue, the value for *‘deleted_at’* field should not be updated manually.

In conclusion, adding *soft delete* increases the complexity of database design as well as the complexity of every single database query and introduces new ways for *bugs* to

show up. If a programmer does not add correct filters at every single query *ghost data* is likely to show. Since the vast majority of queries, especially in the context of this project, are for current data, it is unwise to add this type of *soft delete* capability to every database table, increasing both proneness to errors and resources needed to conduct queries, but instead only to tables where there is a legitimate strong reason for *soft delete* to exist, e.g tables related to *user*. A different approach to prevent data loss, and arguably a better one, is to use a database wide *tombstone* table. In this case queries work with no extra filters and a trigger can be set that copies data from given table to *tombstone* table any time any *delete* is detected. In this case when important data is accidentally deleted, a database administrator can still add it back manually by coping it from the *tombstone* table back into appropriate tables.

As of the release of this document *soft delete* has not been implemented, since the current need for it is very low.

3.2 Data access layer design

It is typically not adviced to write ORM data access methods directly into controllers, as this violates the „dependency inversion principle”, which states that one should depend on abstraction rather than concretion. Breaking this principle forces controllers to fully depend on the concrete implementation of the ORM, thus forcing vast rewrites of controllers if later down the line the method of data access is changed from an ORM to something different. To a degree this also violates the „single-responsibility principle” as in addition to their core responsibility of controlling views controllers also take the responsibility of accessing data from a database access provider.

To get around this problem and to ensure modularity as well as to reduce coupling between ORM and controllers some form of intermediate abstraction is necessary. For chosen the repository design pattern, which is in its core a pattern of designing collections of domain objects with appropriate data access methods for these objects. This itermediate layer releases controllers from the responsibility of where the actual data is coming from and places that responsibility on a separate dedicated component. This ensures that controllers can now depend on repository interfaces rather than concrete implementations and if the concrete implementation is changed later down the line it no longer concerns the controllers.

Another often used pattern to achieve this is the „data access object pattern”, where data is accessed through dedicated data acess objects. The downside of this is that controllers often manipulate data from many different domains and thus require many different data access objects. However, in data access objects transactional methods are often written into the object itself leading to the neccessity of calling „commit” or an equivalent method on every single object. If something does not go right there is a possibility of

broken or partial data being entered into the database. This problem can be solved with using repository design pattern and encapsulating all repositories under a single unit of work, as the repository would act as a collection tracking in-memory changes to its components and the actual commiting of those changes to database is done by a unit of work when the entire transaction was successful. For this reason I have chosen to go with repository and unit of work design.

In my project repositories are currently implemented as an abstraction layer between *Entity Framework DbSets* and controllers. For each domain object a dedicated repository is created, which encapsulates the domain objects *DbSet* and provides methods for accessing, adding and removing records from these sets. These repositories are in turn encapsulated by a unit of work, which provides a *save changes* method, ensuring that all transactional changes to the data are either done all at once or not at all. Each repository derives from a base repository class as well as having the possibility of implementing domain specific methods.

3.3 Business logic layer design

Data access layer or *DAL* for short only abstracts operations related to data querying. In a scalable and possibly very large web application, business logic should reside in its own layer on top of data access layer, this once again enforces the „single responsibility principal” since controllers no longer have to manage both data access and data proccessing.

In the current desing of this application, the business logic layer is divided into smaller *services* which are accessible through the unit of work wrapper around them called *BLL*. Services are once again divided into to separate categories: entity services and general services. Entity services are, as the name implies, related to a single entity and are at the starting phases of the application development nothing more than DAL extensions and abstractions, providing the same methods of data access as DAL. As the application grows, more and more business logic is expected to be added, and the enitity services are expected to be replaced by other more general services.

Ideally *REST* controllers and *ASP.NET* web controllers should both conduct their communication with the database through the business logic layer, since this allows for the application to grow in the sense of business logic without having to mutate the controllers later on.

Business logic layer acts in of itself as a unit of work as well.

3.4 Web service layer

3.4.1 General design

Final client facing layer in this archidecture is the web service layer. Web service layer encompasses both REST controllers as well as regular server side webpage generation controllers. Server side webpage generation controllers for admin views are built using the *model view controller* or *MVC* technology provided by *ASP.NET*, which currently covers workout routine template creation and user account management. The decision why this is done server side rather than client side is that server side webpage generation is inherently less vulnerable to malicious attacks, since it requires as little of the actual code and logic to be reflected out to the client as possible.

API controllers for the user views are done in a representational state transfer archidecture or *REST*, which allows for regular HTTP requests to be parsed and data to be sent out as a response.

3.4.2 Resource security

Since majority of resources in this application are sensitive, meaning they are tied to a specific user, well managed and maintained resource security is of high importance when accessing web controllers. First philosophy at use is that at no point is user specific information reflected out of the system, this protects the system from accidental user infromation leaks. An exception here are the API controllers, which send out JSON web tokens at successful login. JSON web tokens encompass an encrypted hash of user login claims and are verified at every REST request.

3.5 Data transfer between layers

3.5.1 Data transfer objects

Data transfer between database layer, data access layer, business logic layer and web service layer is done by dedicated data access objects or DTOs for every layer. The reason for this is that the data transfered may look drastically different after each layer has processed it, meaning that using a single data trasfer object is unwise.

Data access objects are kept as flat as possible and are preferrably in the confines of a single domain object without nested entities, this provides easier data mapping as well as helps to prevent any issues with the database, where sending down a nested entity may cause records in tables for the nested entities to be deleted that were not intended. Another aspect is that after data has been processed, the DTO coming back to the data access layer might be identical to the one that left, yet they are deep clones of one another, meaning nested entities cause possible problems in database operations such as *update* and *delete*. Entities coming out from the database and upwards through the layers are permitted to have nested entities if and only if those entities are *subentities*, i.e they are in a one-to-many relationship with their parent entity. DTOs are not permitted to contain parent entities, since this is prone to cause cycles in the mapping.

3.5.2 Mapping

Since every layer has its own data trasfer object, the communication between layers needs object mapping. In the current desing each layer knows how to communicate with the one below it and takes in its own data transfer objects from the layer above it, e.g DAL knows how to communicate with domain level through domain entities, and takes in DAL DTO’s from the BLL.

Mapping between different DTO’s is done using dedicated mappers, which are written into a *mapper context* that is specific to each layer and to which they have access to. Each mapper knows how to map entities both ways, e.g a DAL mapper knows how to map Domain to DAL as well as how to map DAL to Domain. Mappers can also access other mappers through *mapper conttext* for potential nested entities that need to be mapped. The reasoning behind using dedicated mappers instead of *automappers* is that *automappers* push error detection into runtime, which is an anti pattern when using a strongly and statically typed language.

Automappers are also much harder to correctly configure and debug for more complex mapping that happens in between some DTOs in DAL to BLL mapping, where, for example, one DTO can be split into two different ones.

4 References

None.

5 Diagrams

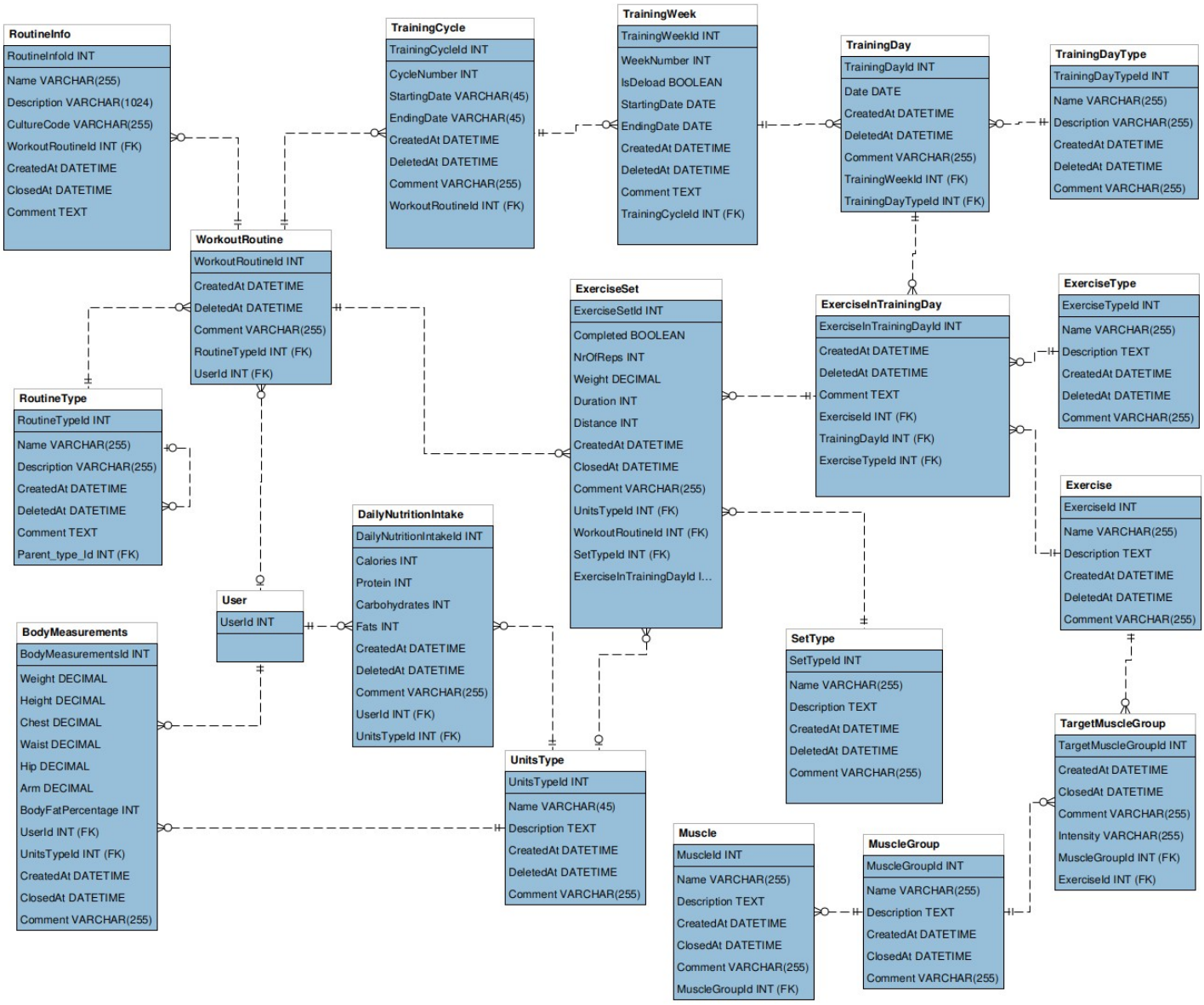

Diagram 1 – ERD Schema