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

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 personal records and overall body composition.

2 Scope of work

2.1 Overview

The scope of work for this project is to build a distributed MVC-based webapplication that allows the user to:

- Create a planned personal workout program based on Jim Wendler’s “5/3/1” routine
- Track personal records for core barbell lifts
- 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
- See a checklist for proper form and instruction for all major barbell exercises

2.2 Project structure

2.2.1 Frontend user interface

This project has two front end user interfaces: desktop and mobile. Desktop user interface is a web user interface developed using *Microsoft*’s *ASP-NET MVC* technology. Mobile user interface is a mobile application using *React native*.

2.2.2 Backend API

Backed for this project isdeveloped using distributed system architecture, where user information is kept in a remote *MySQL* database and accessed and processed by a *.NET Core* application in a webserver.

3 Analysis

3.1 Database design

3.1.1 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.

3.2 Data access 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 this I have 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.

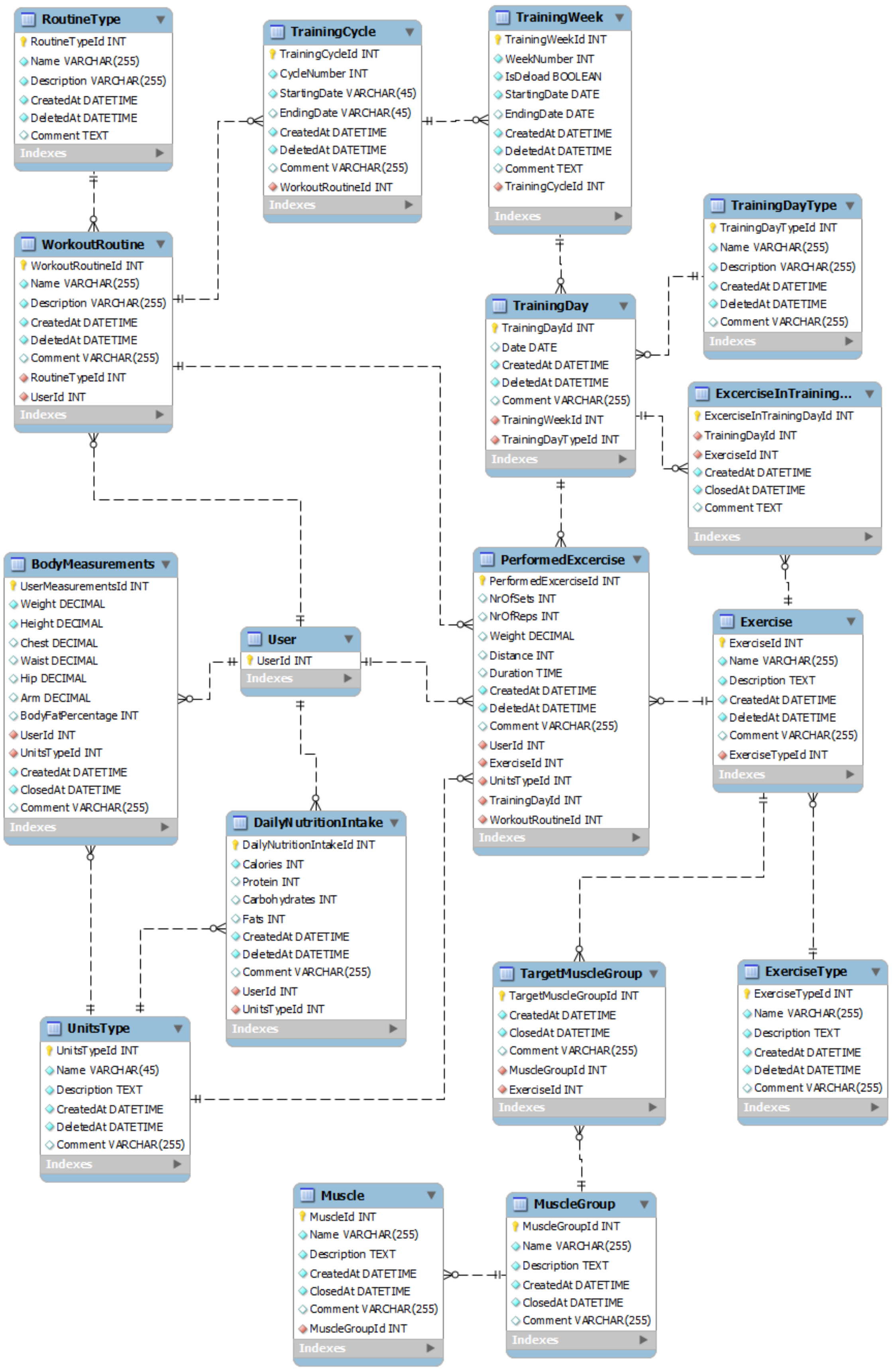
Another often used pattern to achieve this is the „data access object pattern", where data is accessed through dedicated data acess objects. The downside of this is that controllers often manipulate data from many different domains and thus require many different data access objects. However, in data 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.

4 References

None.

5 Diagrams



RoutineType
RoutineTypeId INT
Name VARCHAR(255)
Description VARCHAR(255)
CreatedAt DATETIME
DeletedAt DATETIME
Comment TEXT
Indexes
TrainingCycle
TrainingCycleId INT
CycleNumber INT
StartingDate VARCHAR(45)
EndingDate VARCHAR(45)
CreatedAt DATETIME
DeletedAt DATETIME
Comment VARCHAR(255)
WorkoutRoutineId INT
Indexes
TrainingWeek
TrainingWeekId INT
WeekNumber INT
IsDeload BOOLEAN
StartingDate DATE
EndingDate DATE
CreatedAt DATETIME
DeletedAt DATETIME
Comment TEXT
TrainingCycleId INT
Indexes
TrainingDayType
TrainingDayTypeId INT
Name VARCHAR(255)
Description VARCHAR(255)
CreatedAt DATETIME
DeletedAt DATETIME
Comment VARCHAR(255)
Indexes
WorkoutRoutine
WorkoutRoutineId INT
Name VARCHAR(255)
Description VARCHAR(255)
CreatedAt DATETIME
DeletedAt DATETIME
Comment VARCHAR(255)
RoutineTypeId INT
UserId INT
Indexes
TrainingDay
TrainingDayId INT
Date DATE
CreatedAt DATETIME
DeletedAt DATETIME
Comment VARCHAR(255)
TrainingWeekId INT
TrainingDayTypeId INT
Indexes
ExcerciseInTraining...
ExcerciseInTrainingDayId INT
TrainingDayId INT
ExerciseId INT
CreatedAt DATETIME
ClosedAt DATETIME
Comment TEXT
Indexes
BodyMeasurements
UserMeasurementsId INT
Weight DECIMAL
Height DECIMAL
Chest DECIMAL
Waist DECIMAL
Hip DECIMAL
Arm DECIMAL
BodyFatPercentage INT
UserId INT
UnitsTypeId INT
CreatedAt DATETIME
ClosedAt DATETIME
Comment VARCHAR(255)
Indexes
User
UserId INT
Indexes
PerformedExcercise
PerformedExcerciseId INT
NrOfSets INT
NrOfReps INT
Weight DECIMAL
Distance INT
Duration TIME
CreatedAt DATETIME
DeletedAt DATETIME
Comment VARCHAR(255)
UserId INT
ExerciseId INT
UnitsTypeId INT
TrainingDayId INT
WorkoutRoutineId INT
Indexes
Exercise
ExerciseId INT
Name VARCHAR(255)
Description TEXT
CreatedAt DATETIME
DeletedAt DATETIME
Comment VARCHAR(255)
ExerciseTypeId INT
Indexes
DailyNutritionIntake
DailyNutritionIntakeId INT
Calories INT
Protein INT
Carbohydrates INT
Fats INT
CreatedAt DATETIME
DeletedAt DATETIME
Comment VARCHAR(255)
UserId INT
UnitsTypeId INT
Indexes
TargetMuscleGroup
TargetMuscleGroupId INT
CreatedAt DATETIME
ClosedAt DATETIME
Comment VARCHAR(255)
MuscleGroupId INT
ExerciseId INT
Indexes
ExerciseType
ExerciseTypeId INT
Name VARCHAR(255)
Description TEXT
CreatedAt DATETIME
DeletedAt DATETIME
Comment VARCHAR(255)
Indexes
UnitsType
UnitsTypeId INT
Name VARCHAR(45)
Description TEXT
CreatedAt DATETIME
DeletedAt DATETIME
Comment VARCHAR(255)
Indexes
Muscle
MuscleId INT
Name VARCHAR(255)
Description TEXT
CreatedAt DATETIME
ClosedAt DATETIME
Comment VARCHAR(255)
MuscleGroupId INT
Indexes
MuscleGroup
MuscleGroupId INT
Name VARCHAR(255)
Description TEXT
CreatedAt DATETIME
ClosedAt DATETIME
Comment VARCHAR(255)
Indexes

Diagram 1 – ERD Schema