

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.....	3
2 Scope of work.....	4
2.1 Overview.....	4
2.2 Project structure.....	4
2.2.1 Frontend user interface.....	4
2.2.2 Backend API.....	4
3 Analysis.....	5
3.1 Database design.....	5
3.1.1 Soft delete implementation.....	5
4 References.....	6
5 Diagrams.....	7

# **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

Backend for this project is developed 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 reversible 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

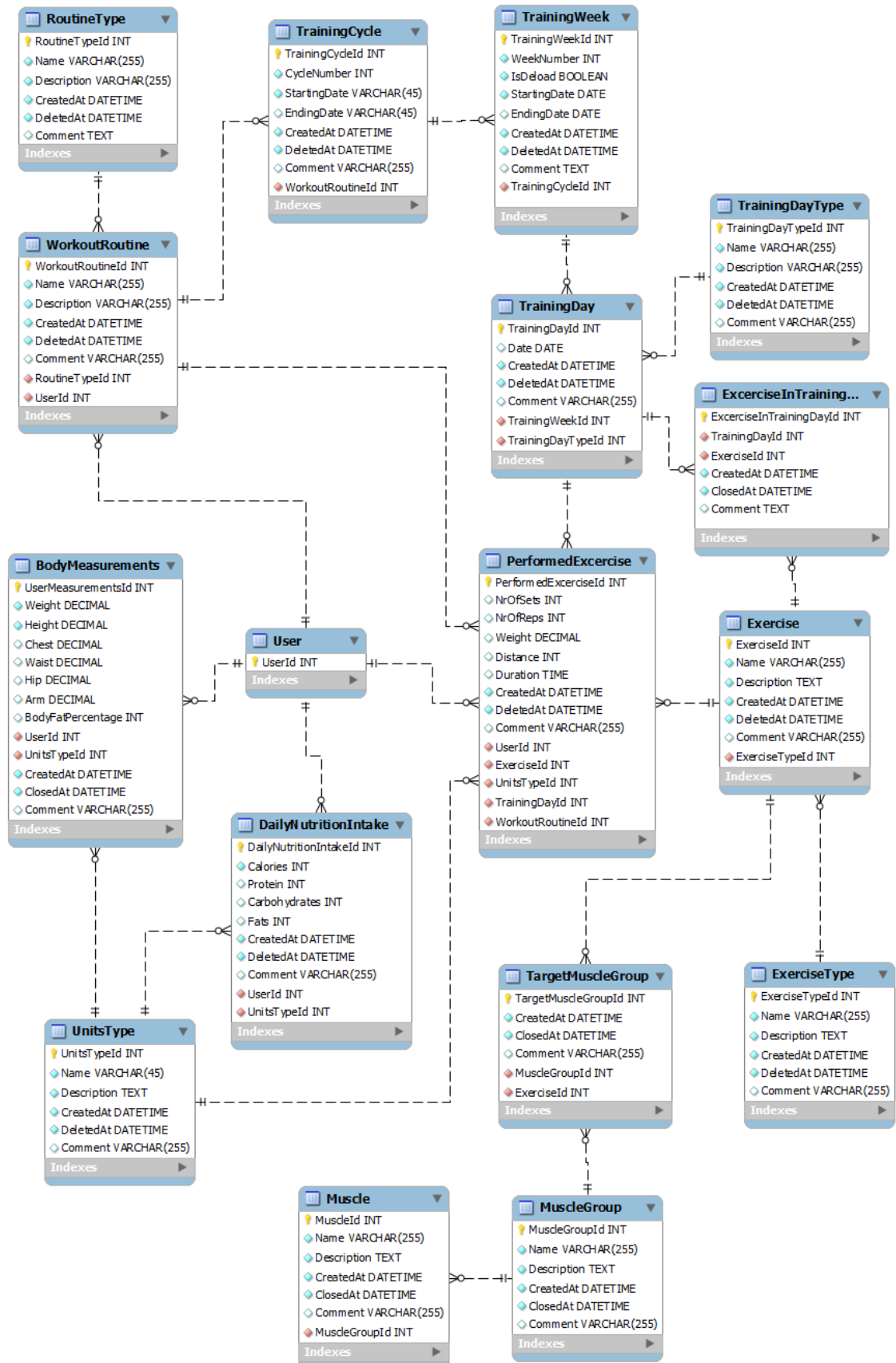
conduct queries, but instead only to tables where there is a legitimate strong reason for *soft delete* to exist, e.g tables related to *user*. A different approach to prevent data loss, and arguably a better one, is to use a database wide *tombstone* table. In this case queries work with no extra filters 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.

## **4 References**

None.



## 5 Diagrams



*Diagram 1 – ERD Schema*