Database Project #2 - Documentation

Author: Ishmael Kwayisi

Project Overview

Our project is a web application that enables users to record information regarding different types of workout sessions (weight lifting, running, etc.). The main distinction of our app compared to other apps is that the user is able to set smart presets based on their personalized time frame preferences.

Team Members

- Ishmael Kwayisi
- Chelsea Alysson Ongjoco
- Kyle Furey

Scope and Requirements

- Functional Requirements
  - Users can add/delete workout sessions
  - Users can query past workout sessions
  - Users can update past workout sessions
  - Users can browse through a list of workouts
  - Users can specify the type of workout they want to record
- Non-functional Requirements
  - Performance

        - App must be able to respond to the user’s request in at least 2 seconds
    - Usability
        - App should be intuitive and easy to navigate for first time users
    - Reliability
        - Application must have 99.9% uptime
    - Maintainability
        - Code contains comments in each file
    - Compatibility
        - Must be able to run on a local device
    - Security
        - User can only enter positive integers
- Out of scope
    - Sharing functionality with contacts
    - Live recording of workouts
    - No video/image storage
    - No map functionality
    - Security features

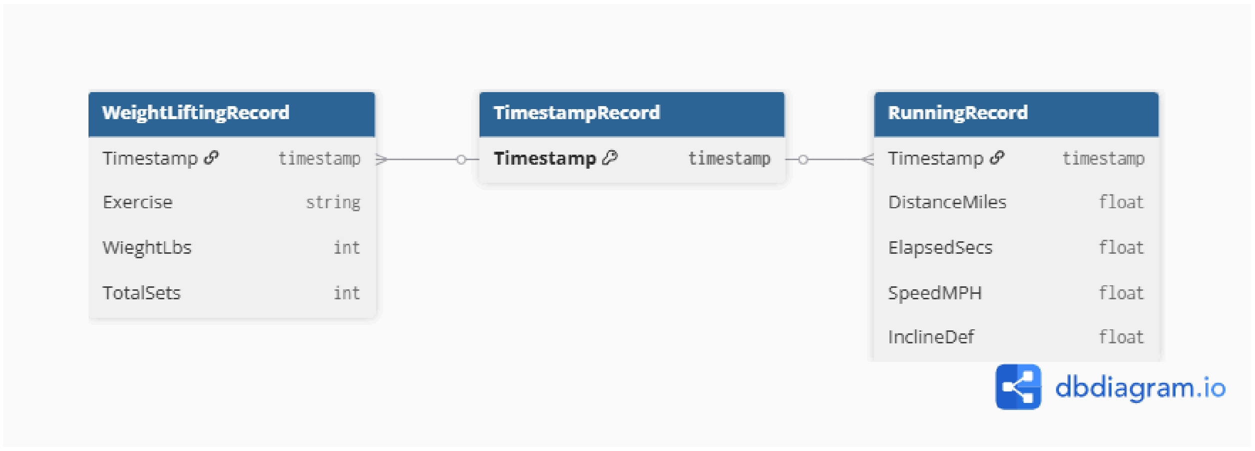
Architecture Design

- Front-end: React + Vite (Lead - Chelsea, Support - Ishmael)
- Back-end: Go (Lead - Kyle, Support - Ishmael)
- Database: MongoDB (Lead - Kyle)

NoSQL Database Selection

For our full-stack app, we chose MongoDB as our NoSQL database. The reason for this is because MongoDB supports “a JSON-like format to store documents” (MongoDB, 2024). Document databases allow us to have full control of our database schema without the need of a database administrator. Another eason for our selection of MongoDB is because it supports an evolving development cycle, such as Agile. Significantly, in a development cycle, objects are naturally mapped through MongoDB through modern programming languages (MongoDB, 2024). Reducing the need for developers to construct complex object-relational mapping (ORMs) and return data becomes more intuitive, similar to native code. Lastly, our choice for choosing MongoDB as our database was influenced by its flexibility in handling evolving data structures. Through MongoDB’s support for document databases, documents are allowed to be embedded to “describe nested structures and easily tolerate variations in data generations of documents” (MongoDB, 2024). Encouraging a resilient repository that doesn’t need to be redesigned whenever change occurs within the database.

Data Model Design



Explanation of Design Decisions

Our database was designed to be a full-stack exercise tracking app where users can log details and track progress of workout sessions. Through the implementation of a database, the user should record multiple workouts, multiple exercises, and have their progress tracked historically. Our database requires the use of three tables, *Weight Lifting, Running,* and *Date.* The separation of these tables is necessary because each table contains their own distinct attributes unique to the table. For instance, the weight lifting table needs a weight attribute to store for users to be aware of the weight they used for the session. While the *Running* table needs a speed or incline attribute to highlight the exercise intensity of a user. If the *Running* table had a weight attribute in the table, it would only confuse and dissatisfy users. While also violating data integrity because in a logical sense, a weight should not be associated with an individual's running session.

As for the relationships between entities, the *Weight Lifting* table has a one-to-many relationship with the *Date* table. While the *Running* table also has a one-to-many relationship with the *Date* table. This is because a single date can have multiple lifting or running entries, while each session belongs to one date. One table that is noticeably not included in our database is a *User* table. The reason behind this is because our focus for development is not on allowing users to create their own login accounts. Thus, we don't particularly need this table in our database, unless we choose to include a user account feature for future implementation of the app.

Added Features/Fixes

- CORS - 2/4/2026

- Recently, CORS errors were only developing on Ishmael’s laptop, but not Chelsea and Kyle’s. To remedy this, we added a function called SetCORS() in the “Client.Go” file to enable the visibility of CORS headers before the first response was written. This is important because it prevents browsers from blocking requests.

- Menu Tabs - 2/5/2026
    - We implemented 2 tab buttons for the user to toggle between. One is for having the user navigate to the logging screen; while the other is to display a summary of past logs from the user.
    - Changes to APP.jsx were made to create the logic to change the information displayed on the screen along with displaying two buttons at the top to toggle between displays. This was designed based on the idea of keeping our app a single page application to encourage simplicity to meet a short deadline.
    - Since the log screen was previously created, in App.jsx we used React hooks to develop the design for the data in the database. Along with utilizing the UseEffect and UseState hook to change the state of the screen depending on user input. Lastly, we chose to locate the buttons at the top of the screen to replicate tabs you’d find on a home screen page.

- Handling Invalid Inputs - 2/5/2026
    - We noticed the user is able to input negative values into the log and submit it with no errors. However, since we want to display accurate and logical data we needed

a way to handle these invalid inputs. Our plan was to make edits in both the frontend and backend.

  - In the App.jsx file, the plan was to add logic that ensures a scenario where if the user enters and submits a negative value, the output is changed to a 0. Else, the output displays regularly as a float value.
  - Then in the "Client.Go" file, we add code to produce an 400 error bad request message. The purpose of this was to ultimately prevent the user from submitting any negative values at all and enforce the accuracy and reliability in our database.
  - Lastly, after our work in the backend, we added HTML to display text informing the user that their input is invalid in the scenario. We also added some CSS to stylize the text for intended rhetoric.

Setup and Installation

For a user to use our app locally, the use of two terminals will be required. One for the client and the other for the server. For the client, the user needs to run:

- npm install
- npm run dev

While the server, the user needs to run:

- go run main.go

Usage Guide

1. User enters the web application
2. User is greeted with screen prompting their choice of exercise to record

3. Depending on the exercise type chosen, the user will be able to insert details about the workout session
4. Once the user is finished, the details of the session will be added to a database
5. The user is transferred to a menu displaying their most recent workout session
6. The user will then have several options to choose from this point
    a. Edit any added session
    b. Add a new workout session
    c. Delete any session present in the database
    d. Query for workout sessions

References

dbdiagram.io - Database Relationship Diagrams Design Tool. (n.d.). Dbdiagram.io. Retrieved February 1, 2026, from https://dbdiagram.io/home

MongoDB. (2024). *Why Use MongoDB And When To Use It?* MongoDB. https://www.mongodb.com/resources/products/fundamentals/why-use-mongodb