## DESCRIPTION

MyHealthTracker is a web-based health management application that allows users to track medical test results, monitor trends, and assess health risks using data analytics and machine learning

By Danielle Hodaya Shrem and Shiran Levi

# MY HEALTH TREACKER

Software Documentation

My Health Tracker

## Table of Contents

# Part 1: Datasets

## Main Dataset - Health Checkup Result dataset -

The dataset originates from Kaggle's Health Checkup Result dataset and contains health examination records collected from 2002 to 2020. Due to memory constraints and query performance considerations, we only utilized data from **2015 onwards**.

The dataset consists of multiple CSV files categorized by year. We processed and loaded the relevant yearly files into the **myhealthtracker** database.

### Used and Unused Data

**Unused Columns**

To optimize storage and improve query efficiency, we excluded non-essential columns related to vision and dental health, including:

- SIGHT_LEFT, SIGHT_RIGHT (Vision-related measurements)
- HCHK_CE_IN (Oral examination participation)
- CRS_YN (Presence of dental caries)
- TTH_MSS_YN (Missing teeth)
- ODT_TRB_YN (Dental abrasion)
- WSDM_DIS_YN (Wisdom teeth presence)
- TTR_YN (Presence of calculus)

**Used Columns**

We primarily focused on demographic, lifestyle, and medical test data. Based on the Python scripts (insert_to_DB.py and API code), the following columns were used:

Demographic Information

- YEAR – The year of data collection.
- IDV_ID – Unique identifier for the examinee.
- SEX – Gender (1: Male, 2: Female).
- AGE_GROUP – Age group classification.
- HEIGHT, WEIGHT – Physical characteristics.

Medical Tests

- **Blood Pressure**: BP_HIGH, BP_LWST
- **Blood Glucose**: BLDS
- **Cholesterol & Lipids**: TOT_CHOLE, TRIGLYCERIDE, HDL_CHOLE, LDL_CHOLE
- **Kidney Function**: CREATININE

- **Hemoglobin**: HMG
- **Urinary Protein**: OLIG_PROTE_CD
- **Liver Function Tests**: SGOT_AST, SGPT_ALT, GAMMA_GTP

Lifestyle Factors

- SMK_STAT – Smoking status (1: Non-smoker, 2: Quit smoking, 3: Smoker).
- DRK_YN – Drinking status (0: Non-drinker, 1: Drinker).

Metadata

- DATE – Date when the record was created.

## Data Insertion and Update Process

The ID, along with age, weight, height, and other demographic data, was inserted into the Users table.
Test results, along with their corresponding dates, were stored in the User_Tests table.
Lifestyle-related information was recorded in the Life_Style table.

The data insertion process was divided into two main stages: **initial data insertion** and **annual updates**.

**Initial Data Insertion**

The first stage involved loading the initial dataset (`/data/2015.csv`) into the database. Since this dataset served as the foundation for user records, several preprocessing steps were applied to ensure data consistency and completeness:

**1. Lifestyle Data Generation**

Since the dataset lacked lifestyle attributes such as physical activity, marital status, education level, work, dietary habits, and sleep patterns, random values were assigned to complete the dataset.

**2. Password Generation**

Each user was assigned a username based on their ID from the dataset.
Passwords were generated in the format: "pass" + ID (e.g., pass12345).

All code sections related to password and lifestyle data generation can be found in `/backend/insert_to_DB.py`.

**Annual Updates**

After the initial data insertion, new datasets from subsequent years were processed to update user records. Unlike the initial phase, these updates focused solely on adding new test results and updating user age.

Lifestyle attributes and passwords remained unchanged, as they were assumed to be static over time. Instead, user ages were recalculated based on their age group, using a predefined mapping to determine a randomized real age within the given range. This updated age was then reflected in the Users table.

Additionally, new test results from each year were inserted into the User_Tests table, ensuring that historical medical data was preserved for longitudinal analysis.

To optimize database performance, data was processed in chunks of 10,000 rows, reducing query execution time and improving system efficiency.

The corresponding code located in **/backend/update_DB.py.**

**Data Cleaning and Preparation**

During data insertion, several preprocessing steps were applied to handle date formats, missing values, and the removal of non-numerical data.

- **Date Format Handling:** DATE fields were standardized using the function 'is_valid_date' in **/backend/update_DB.py.**
- **Non-Numeric Value Conversion:** Smoking (SMK_STAT) and drinking (DRK_YN) values were transformed from 'N'/'Y' to numeric values (0/1) for consistency.

Once all data was inserted, the last 500,000 users were **removed from the database** to reduce the dataset size and improve query efficiency, retaining only the highest IDV_ID values.

The deletion process was executed after all data had been uploaded to the database. First, a temporary table was created to store the IDs of users marked for deletion. Then, User records were removed from the User_Tests, Life_Style, and Users tables.

The deletion logic is implemented in **/backend/delete_users.py**. After the removal process, database optimization was performed using **/backend/optimizeTable.py** to enhance performance.

## Additional Datasets –

We used other datasets in machine learning models to predict risks for heart diseases, diabetes, stroke and depression.

- **Heart Disease Dataset**
  **Source:** Heart Disease Dataset
  This dataset contains patient medical records related to heart disease, including blood pressure, cholesterol levels, blood sugar levels, and demographics. The target variable indicates whether a patient has heart disease or not.

  Columns Used:

  - age – Patient's age.
  - sex – Gender (1 = Male, 0 = Female).
  - trestbps – Resting blood pressure (mm Hg).
  - chol – Serum cholesterol (mg/dL).
  - fbs – Fasting blood sugar (1 = high, 0 = normal).
  - target – Indicates the presence (1) or absence (0) of heart disease.

Columns Not Used: cp (chest pain type), restecg (resting electrocardiographic results), thalach (maximum heart rate achieved), exang (exercise-induced angina), oldpeak, slope, ca, thal (various ECG and stress test results).

These unused columns relate to ECG and stress tests, which were not part of our available user data.

- **Diabetes Health Indicators Dataset**

**Source:** Diabetes Health Indicators Dataset
This dataset includes health indicators relevant to diabetes risk, such as BMI, physical activity, blood pressure, cholesterol levels, smoking, and alcohol consumption. The target variable indicates whether a patient has diabetes.

Columns Used:

- HighBP – High blood pressure status (0 = No, 1 = Yes).
- HighChol – High cholesterol status (0 = No, 1 = Yes).
- BMI – Body Mass Index.
- Smoker – Whether the patient has ever smoked.
- PhysActivity – Engages in physical activity (0 = No, 1 = Yes).
- HvyAlcoholConsump – Heavy alcohol consumption (0 = No, 1 = Yes).
- Sex – Gender (Mapped to match our system's values).
- Age – Age group (Mapped to our system's 13-level scale).
- Education – Level of education.
- Diabetes_binary – Target variable (0 = No Diabetes, 1 = Diabetes).

Columns Not Used: Income, general health, mental health, and physical health indicators were excluded to fit our available data.

- **Stroke Prediction Dataset**

**Source:** Stroke Prediction Dataset
This dataset includes patient attributes related to stroke risk, such as age, hypertension, glucose levels, BMI, smoking status, and medical history. The target variable indicates whether a patient has experienced a stroke.

Columns Used:

- gender – Mapped to numerical values (1 = Male, 2 = Female).
- age – Patient's age.
- hypertension – History of high blood pressure (0 = No, 1 = Yes).
- ever_married – Marital status (Mapped to numerical values).
- avg_glucose_level – Average blood glucose level.
- bmi – Body Mass Index.

- smoking_status – Smoking habits (Mapped to numerical values).
- stroke – Target variable (0 = No Stroke, 1 = Stroke).

Columns Not Used: Residence type, employment type, and heart disease status.

❖ **Depression Dataset**
**Source:** Depression Dataset
This dataset contains demographic and lifestyle factors associated with depression risk, including marital status, education level, physical activity, smoking, alcohol consumption, sleep patterns, and history of mental illness.

Columns Used:

- Age – Age of the individual.
- Marital Status – Single, Married, Divorced, Widowed.
- Education Level – High School, Associate Degree, Bachelor's, Master's, PhD.
- Number of Children – Number of children in the household.
- Smoking Status – Smoker, Former Smoker, Non-Smoker.
- Physical Activity Level – Sedentary, Moderate, Active.
- Employment Status – Unemployed, Employed.
- Alcohol Consumption – Low, Moderate, High.
- Dietary Habits – Healthy, Moderate, Unhealthy.
- Sleep Patterns – Good, Fair, Poor.
- History of Mental Illness – Target variable (0 = No, 1 = Yes).

Columns Not Used: Income level, social support, and daily stress levels were excluded since they were not part of our application's available user data.

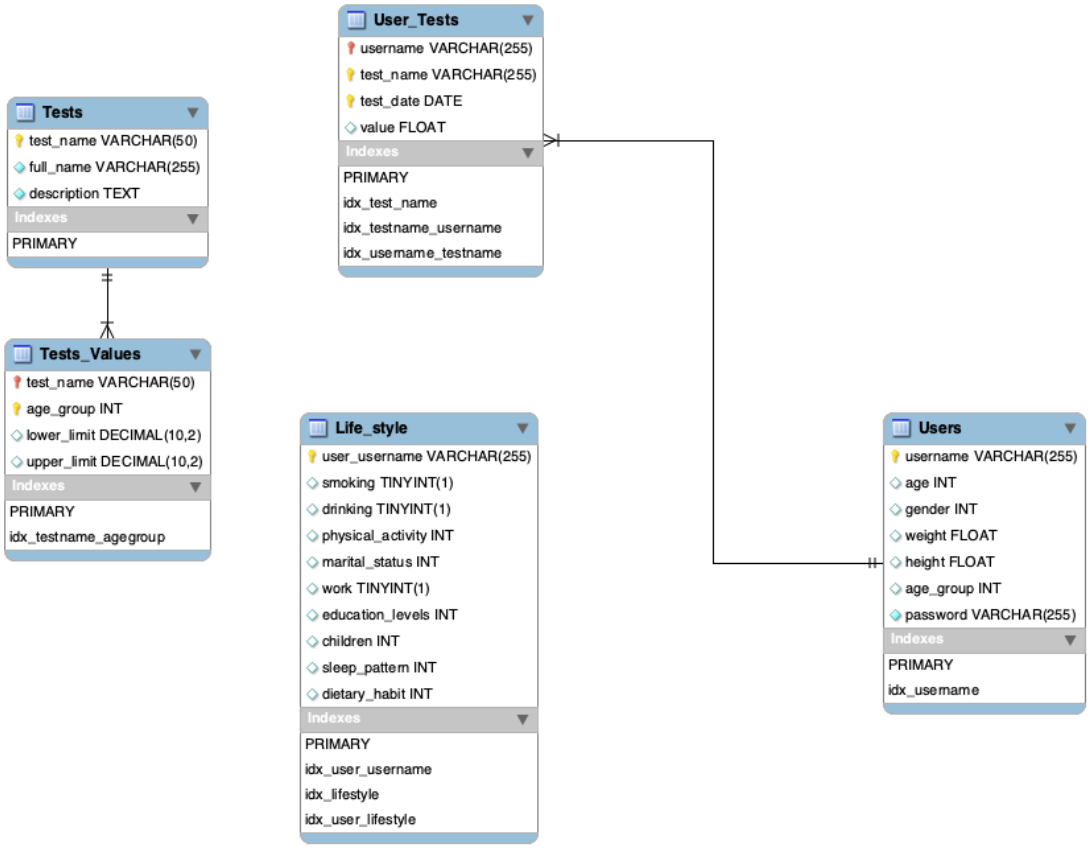
## Sources Used for Medical Reference Data -

1. **CDC – Centers for Disease Control and Prevention**
   Provides medical guidelines and reference values for blood pressure, cholesterol, and blood sugar levels.
2. **WHO – World Health Organization**
   Offers global health guidelines, including recommended reference ranges for various medical tests.
3. **Schneider Children's Medical Center of Israel**
   This source provides pediatric blood test reference values based on age for childrens.
4. **Tel Aviv Sourasky Medical Center (Ichilov)**
   A guide for interpreting complete blood count (CBC) and other laboratory tests.

The data from these sources entered to test_values table on the DB.

# Part 2: Database Scheme

## Final Scheme




## Tables Overview

1. **Users**: Stores demographic and personal health information, including username, age, gender, weight, height, and age group.
2. **Life_style**: Records lifestyle-related factors such as smoking, drinking, physical activity, education levels, and other lifestyle habits.
3. **User_Tests:** Contains test results for individual users, including test name, test date, and value.
4. **Tests**: Provides metadata for different test types, including test_name, full name, and description.
5. **Tests_Values:** Defines reference ranges for different test results based on age group, with lower limit and upper limit values.

## Tables and Relationships

### 1. Users

- **Primary Key**: username
- **Fields**: username, password, age, height, weight, gender, age_group

- **Indexes**:
  - idx_username: Optimizes user lookup queries.
- **Relationships**:
  - One-to-one with Life_style (username → user_username in Life_style)
  - One-to-many with User_Tests (username → username in User_Tests)

**2. Life_style**

- **Primary Key**: user_username
- **Fields**: user_username, smoking, drinking, physical_activity, marital_status, work, education_levels, children, sleep_pattern, dietary_habit
- **Indexes**:
  - idx_lifestyle: Optimizes lifestyle-based queries.
- **Relationships**:
  - One-to-one with Users (user_username → username in Users)

**3. User_Tests**

- **Primary Key**: (username, test_name, test_date)
- **Fields**: username, test_name, test_date, value
- **Indexes**:
  - idx_test_name: Optimizes queries fetching test details.
  - idx_testname_username: Optimizes fetching tests per user.
  - idx_username_testname: Helps with ordering user tests by name.
- **Relationships**:
  - Many-to-one with Users (username → username in Users)
  - Many-to-one with Tests (test_name → test_name in Tests)

**4. Tests**

- **Primary Key:** test_name
- **Fields**: test_name, full_name, description
- **Relationships**:
  - One-to-many with User_Tests (test_name → test_name in User_Tests)
  - One-to-many with Tests_Values (test_name → test_name in Tests_Values)

**5. Tests_Values**

- **Primary Key**: (test_name, age_group)
- **Fields**: test_name, age_group, lower_limit, upper_limit
- **Indexes**:
  - idx_testname_agegroup: Optimizes fetching test limit values by test and age group.
- **Relationships:**
  - Many-to-one with Tests (test_name → test_name in Tests)

## Indexes and Query Optimization

Indexes are strategically placed to optimize specific queries:

- **User lookup (Users)**
  - idx_username speeds up login queries and profile retrieval.
- **Fetching user test history (User_Tests)**
  - idx_testname_username and idx_username_testname optimize test retrieval and sorting.
- **Comparing test results (Tests_Values)**
  - idx_testname_agegroup improves efficiency in retrieving reference ranges.

## Part 3: Queries Explanation

**1. Get User's Data**

Location: **`/backend/auth_api.py`** function *'get_user_data()'*

Query –

```
SELECT u.username, u.age, u.gender, u.weight, u.height,
u.age_group, l.smoking, l.drinking, l.physical_activity,
l.education_levels

FROM Users u

LEFT JOIN Life_style l ON u.username = l.user_username

WHERE u.username = %s;
```

This query retrieves a user's demographic and lifestyle information by selecting data from the Users table and joining it with the Life_style table using a LEFT JOIN. It filters the results based on the given username. The query used to present user's data and for further analysis like BMI calculation and comperisons.

**2. Add Test**

Location: **/backend/auth_api.py** function *'add_test()'*

Query -

```
INSERT INTO User_Tests (username, test_name,
test_date, value)

VALUES (%s, %s, %s, %s)
```

This query inserts a new test result into the User_Tests table, storing details about a user's medical test, including the username, test_name, test_date, and value. It ensures that all required fields are provided before executing the insertion.

Example –

```
INSERT INTO User_Tests (username, test_name,
test_date, value)

VALUES ('john_doe', 'BP_HIGH', '2024-02-03', 130);
```

**3. Get User's Test History**

Location: **/backend/auth_api.py** function *'get_user_tests()'*

Query -

```
SELECT * FROM User_Tests

WHERE username = %s

ORDER BY test date DESC
```

This query retrieves all medical test records for a given user from the User_Tests table. It filters the results using the username parameter and orders them by test_date in descending order, ensuring the most recent tests appear first. The index on username speeds up retrieval, and the order by test_date DESC ensures efficient sorting for time-based queries. This query is essential for displaying a user's medical history, tracking test trends, and enabling health analysis based on past results.

**4. Compare User Test Results with Similar Users**

Location: **/backend/auth_api.py** function *'compare_tests()'*

**Query 1** – Create a Temporary Table for Similar Users

```
CREATE TEMPORARY TABLE temp_similar_users AS

SELECT username

FROM Users

JOIN Life_style ON Users.username = Life_style.user_username

WHERE age_group = %s AND gender = %s AND smoking = %s    AND
drinking = %s AND physical_activity = %s

LIMIT 1000;
```

This query finds up to 1000 users with similar lifestyle and demographic characteristics and stores them in a temporary table. The indexes on Users.username and Life_style.user_username ensure efficient filtering and joining. To optimize query performance, we limit the comparison to a maximum of 1000 similar users. This ensures efficient execution while maintaining meaningful statistical analysis.

**Query 2** - Retrieve the User's Own Test Results

```
SELECT test_name, value AS user_value

FROM User_Tests

WHERE username = %s;
```

Fetches all test results for the given user for direct comparison.

**Query 3** - Generate Histogram Data for Similar Users

```
SELECT test_name, FLOOR(value / 10) * 10 AS bin,

    COUNT(*) AS frequency

FROM User_Tests

WHERE username IN

(SELECT username

FROM temp_similar_users)

GROUP BY test_name, bin

ORDER BY test_name, bin;
```

This query groups test values of similar users into bins of 10-unit ranges and counts how many users fall into each bin. Helps analyze how the user's results compare to their peer group. This data is essential to create a histogram.

**Query 4 -** Drop the Temporary Table

```
DROP TEMPORARY TABLE IF EXISTS temp_similar_users;
```

Cleans up the temporary data to free memory and maintain database efficiency.

<u>Indexes That Optimize Performance:</u>

- **PRIMARY KEY (username) on Users** → Ensures fast lookups for finding similar users.
- **Index on Life_style.user_username** → Speeds up filtering lifestyle-related data.
- **Index on User_Tests.username** → Optimizes fetching test results for the user.
- **Index on User_Tests.test_name** → Enhances performance when grouping histogram data.

The analysis uses a temporary table to efficiently handle comparisons in a dataset where the User_Test table contains approximately 35 million records. Without a temporary table, filtering similar users directly in every query would require scanning a massive dataset repeatedly, significantly slowing down execution.

By first creating a temporary table that holds up to 1000 similar users, we reduce the search space for subsequent queries. This optimization allows the database engine to work with a much smaller subset of data, making joins and aggregations (such as histograms) significantly faster.

5. **Retrieve Available Tests**

Location: **/backend/auth_api.py** function *'get_tests()'*

Query -

```
SELECT test_name, full_name, description

FROM Tests;
```

This query retrieves all available medical tests from the Tests table, including their name, full name, and description. It provides a list of test options that users can select when recording their medical data.

6. **Retrieve Test Limits by Age Group**

Location: **/backend/auth_api.py** function *'get_all_test_limits()'*

**Query 1** – Fetch the User's Age Group

```
SELECT age_group

FROM Users

WHERE username = %s;
```

This query retrieves the user's age group, which is necessary for selecting the correct test limits.

**Query 2**- Retrieve Test Limits for the Age Group

```
SELECT test_name, lower_limit, upper_limit

FROM Tests_Values

WHERE age_group = %s;
```

This query fetches the normal value ranges for all medical tests relevant to the user's age group. Optimized using INDEX (age_group, test_name) to ensure quick filtering and lookup.
The query plays a role in personalizing medical insights, allowing users to better understand their health status relative to established medical guidelines.

**7. Authenticate User Login**

Location: **`/backend/auth_api.py`** function *'login()'*

Query -

```
SELECT * FROM Users

WHERE username = %s AND password = %s;
```

The query searches for a user whose username and password match the given input. If a matching record is found, the system considers the login successful and returns the user's details.

**8. Update User Personal Information**

Location: **`/backend/auth_api.py`** function *'update_user_info()'*

Query –

```
UPDATE Users

SET age = %s, height = %s, weight = %s

WHERE username = %s;
```

The query takes in new values for age, height, and weight, filtering by username to ensure only the correct user is updated. If the update succeeds, the database commits the changes and returns a success message.

**9. Register a New User**

Location: **`/backend/auth_api.py`** function *'signup()'*

**Query 1 –**

```
SELECT username FROM Users WHERE username = %s;
```

Ensures that the username is not already taken.

**Query 2 -**

```
INSERT INTO Users (username, password, height,
weight, age, age_group, gender)

VALUES (%s, %s, %s, %s, %s, %s, %s);
```

Adds the user's details to the database, ensuring all required fields are set.

Example -

```
INSERT INTO Users (username, password, height, weight,
age, age_group, gender)

VALUES ('john_doe', 'securepass123', 180, 75, 30, 3, 1);
```

**10. Retrieve Health Alerts for a User**

Location: **`/backend/auth_api.py`** function *'get_health_alerts()'*

Query -

```
SELECT t.test_name, ts.full_name, t.test_date, t.value,
v.lower_limit, v.upper_limit

FROM User_Tests t

JOIN Tests_Values v ON t.test_name = v.test_name

JOIN Tests ts ON t.test_name = ts.test_name

WHERE t.username = %s AND v.age_group = %s

AND t.test_date = (SELECT MAX(t2.test_date)

       FROM User_Tests t2

      WHERE t2.username = t.username
      AND t2.test_name = t.test_name)

ORDER BY t.test_date DESC;
```

The query retrieves a user's most recent test results and compares them against predefined reference limits for their age group. It joins three tables: User_Tests, which stores individual test results; Tests_Values, which defines the lower and upper limits for each test based on age group; and Tests, which provides additional metadata about each test. The filtering condition t.username = %s AND v.age_group = %s ensures that only the specific user's test results are retrieved while matching the correct reference values for their age group.

To ensure that only the latest test entry for each test is considered, the query includes a subquery that scans all previous test records for the user and selects the most recent entry for each test. It ensures that outdated test results do not interfere with current health assessments. The final output includes the test name, the recorded value, the date of the test, and the corresponding reference limits.

To optimize performance, this query benefits from several indexes:

- **An index on username, test_name, test_date in User_Tests** helps the database efficiently retrieve the latest test result.
- **An index on test_name, age_group in Tests_Values** speeds up the lookup of reference limits.

This query is used to identify abnormal test results by comparing a user's most recent medical test values against established reference ranges for their age group.

**11. Retrieve User Test Summary**

Location: **/backend/auth_api.py** function *'get_user_test_summary()'*

Query –

```
SELECT t.test_name, ts.full_name, t.test_date, t.value,
v.lower_limit, v.upper_limit

FROM User_Tests t

JOIN Tests_Values v ON t.test_name = v.test_name

JOIN Tests ts ON t.test_name = ts.test_name

WHERE t.username = %s AND v.age_group = %s

AND t.test_date = (SELECT MAX(t2.test_date)

                   FROM User_Tests t2

WHERE t2.username = t.username
AND t2.test_name = t.test_name)

ORDER BY t.test_date DESC;
```

The query fetches the latest test results from the User_Tests table, ensuring that only the most recent entry for each test is included. By joining with Tests_Values, it retrieves the corresponding lower and upper limits for the test based on the user's age group. Additionally, it joins with the Tests table to provide a full test name for better readability.

The query results help to determine if a test result is abnormal, it checks whether the recorded value falls outside the reference range. It also calculates whether a test result is overdue by checking if it was recorded more than a year ago. The query is optimized using indexes on username, test_name, and test_date, which significantly improve retrieval performance.

**12. Insert or Update Lifestyle Information**

Location: **/backend/auth_api.py** function *'update_lifestyle()'*

Query –

```
INSERT INTO Life_style (user_username, marital_status,
education_levels, children, physical_activity, work,
dietary_habit, sleep_pattern, drinking, smoking)

VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)

ON DUPLICATE KEY UPDATE

    marital_status = VALUES(marital_status),

    education_levels = VALUES(education_levels),

    children = VALUES(children),

    physical_activity = VALUES(physical_activity),

    work = VALUES(work),

    dietary_habit = VALUES(dietary_habit),

    sleep_pattern = VALUES(sleep_pattern),

    drinking = VALUES(drinking), smoking = VALUES(smoking);
```

This query updates or inserts a user's lifestyle information in the Life_style table. It attempts to insert a new record with details such as marital status, education level, number of children, physical activity, work status, dietary habits, sleep patterns, drinking, and smoking habits. If the user_username already exists in the table, it updates the existing record using ON DUPLICATE KEY UPDATE, ensuring that no duplicate entries are created while keeping the data up to date.

**13. Retrieve Lifestyle Information**

Location: **`/backend/auth_api.py`** function *'get_lifestyle()'*

Query -

```
SELECT * FROM Life_style

WHERE user_username = %s;
```

This query fetches a user's lifestyle details from the Life_style table based on their username.

## SQL Queries Used for Predictive Models

Each predictive model (diabetes, heart disease, stroke, and depression) requires specific features based on medical test results, lifestyle attributes, and demographic data.
To retrieve the necessary data for each model, we utilized SQL queries that extract relevant information from the database, ensuring that the model has all required inputs for accurate predictions.

**Diabetes Model**

Location: **`/backend/auth_api.py`** function *'predict_diabetes()'*

1. **Fetching User Personal and Lifestyle Data**

```
SELECT Users.age, Users.height, Users.weight,

    Users.gender,Life_style.smoking, Life_style.drinking,

 Life_style.physical_activity, Life_style.education_levels

FROM Users

LEFT JOIN Life_style

ON Users.username = Life_style.user_username

WHERE Users.username = %s
```

2. **Fetching Required Medical Test Results**

```
SELECT
    test_name, value
FROM User_Tests
WHERE username = %s AND test_name IN ('BP_HIGH',
'BP_LWST', 'TOT_CHOLE')
```

The two SQL queries extract the relevant user data required for the predictive diabetes model based on the provided username. The first query retrieves demographic and lifestyle attributes from the Users and Life_Style tables, while the second query fetches specific medical test results from the User_Tests table. These queries ensure that the model receives all necessary inputs for accurate risk prediction and filtering only the required information.

** The queries used for other models are built on the same foundation, fetching only the necessary information for each specific model. The only difference is the columns returned by the query.

**Stroke model**

Location: **`/backend/auth_api.py`** function *'predict_stroke()'*

1. **Fetching User Personal and Lifestyle Data**

```
SELECT Users.gender, Users.age, Users.height,
    Users.weight, Life_style.smoking,
    Life_style.marital_status
FROM Users
LEFT JOIN Life_style
ON Users.username = Life_style.user_username
WHERE Users.username = %s
```

2. **Fetching Required Medical Test Results**

```
SELECT test_name, value
FROM User_Tests
WHERE username = %s AND test_name IN ('BP_HIGH',
'BP_LWST', 'BLDS')
```

**Heart Disease model**

Location: **`/backend/auth_api.py`** function *'predict_heart_diesease()'*

**Fetching User Personal and Lifestyle Data, and Required Medical Test Results**

```
SELECT Users.age, Users.gender, Users.height,
Users.weight, Life_style.smoking,
User_Tests.test_name, User_Tests.value

FROM Users

LEFT JOIN Life_style

ON Users.username = Life_style.user_username

LEFT JOIN User_Tests

ON Users.username = User_Tests.username

WHERE Users.username = %s AND User_Tests.test_name
IN ('BP_HIGH', 'BP_LWST', 'TOT_CHOLE', 'BLDS')
```

**Depression model**

Location: **`/backend/auth_api.py`** function *'predict_depression()'*

**Fetching User Personal and Lifestyle Data**

```
SELECT age, marital_status, education_levels,
children, smoking,

    physical_activity, work, drinking,
dietary_habit, sleep_pattern

FROM Users

LEFT JOIN Life_style

ON Users.username = Life_style.user_username

WHERE Users.username = %s
```

The depression model relies solely on age and lifestyle data, meaning it does not require medical test results.

# Part 4: Code Files Documentation

## Server-Side Code (`/backend/` folder)

The /backend/ folder contains **Python** files responsible for handling the server-side operations.

### 1. User Authentication & Health Data API

The **`auth_api.py`** file serves as the backend API for user authentication, health data management, and machine learning-based predictions. It allows users to register, log in, update personal and health data, retrieve test results, compare them with similar users, and receive risk predictions for various diseases. The API also generates alerts for abnormal test values and ensures efficient retrieval of medical information for personalized health tracking.

User Management –

- **/api/signup (POST)**: This endpoint registers a new user by storing details such as username, password, age, weight, age group, and gender. It also verifies if the username already exists before inserting the record into the database.
- **/api/login (POST)**: This endpoint handles user authentication. It checks if the provided username and password match an existing record in the Users table. If successful, it returns user details; otherwise, it returns an error message.

User Data Retrieval –

- **/api/user_data (GET)**: This endpoint fetches user information, including age, gender, weight, height, age group, smoking status, drinking habits, physical activity, and education level. It also calculates the user's BMI if height and weight are available.
- **/api/get_lifestyle_info (GET)**: This retrieves the user's lifestyle information, such as marital status, education level, number of children, physical activity, work status, dietary habits, sleep pattern, smoking, and drinking status.
- **/api/update_user_info (POST)**: Allows users to update their personal details, including age, height, and weight.
- **/api/update_lifestyle_info (POST)**: Updates the user's lifestyle attributes such as marital status, education, children, smoking, drinking, and physical activity.

Health Test Management –

- **/api/add_test (POST)**: Adds a new health test result for a user. It requires the username, test name, test date, and value.
- **/api/get_user_tests (GET)**: Retrieves all test results associated with a user, ordered by the test date.
- **/api/get_tests (GET)**: Returns a list of available test names along with their descriptions from the Tests table.
- **/api/get_all_test_limits (GET)**: Fetches the lower and upper limit values for all tests based on the user's age group.

Data Comparison & Analysis -

- **/api/compare_tests (GET)**: Compares a user's test results with other users who share similar lifestyle attributes (smoking, drinking, physical activity, and education level). It returns statistical insights such as the average, standard deviation, minimum, and maximum values for each test.
- **/api/user_health_alerts (GET)**: Generates alerts when the user's test values exceed predefined health limits. The endpoint retrieves the most recent test results and checks if they fall outside the normal range.
- **/api/user_test_summary (GET):** Summarizes the user's latest test results, indicating whether any values are out of range or if the test is overdue (older than one year).

Machine Learning Predictions -

- **/api/predict_heart_disease (GET)**: Predicts the user's risk of heart disease based on factors like blood pressure, cholesterol levels, and glucose levels. Uses a pre-trained machine learning model.
- **/api/predict_stroke (GET)**: Predicts stroke risk based on user data such as age, gender, hypertension, smoking status, glucose levels, and BMI.
- **/api/predict_diabetes (GET)**: Assesses diabetes risk using indicators like blood pressure, cholesterol, BMI, and lifestyle factors. The endpoint checks for missing test results before running the prediction.
- **/api/predict_depression (GET)**: Evaluates the likelihood of depression based on user-reported factors like marital status, education, smoking, alcohol consumption, dietary habits, and sleep patterns.

All prediction endpoints are connected to the prediction models located in the `/models/` folder. Model explanations are discussed in the next section.

## 2. Optimization and Database Maintenance Files

These files handle database optimization, including indexing, deleting users, and optimizing table performance.

- **`create_index.py`** (Database Index Optimization) - This script enhances query performance by creating indexes on key columns in the database. It defines indexes for several tables: the Users table gets an index on the username column to enable quick lookups, the User_Tests table has an index on username and test_name to optimize searches for specific test results, the Life_style table gets an index on user_username, smoking, drinking, physical_activity, and education_levels to improve filtering based on lifestyle factors, and the Tests_Values table is indexed on test_name and age_group for efficient retrieval of test value limits. Each query is executed inside a loop, ensuring all indexes are created efficiently, with changes committed after each execution.

- **`delete_users.py`** (Batch User Deletion) - This script manages user deletions in a controlled and efficient manner by processing users in batches. It first creates a temporary table called TempUsersToDelete, which stores a subset of users marked for deletion. The script then selects a batch of users from the UsersToDelete table and removes their corresponding records from User_Tests, Life_style, and Users. After deleting the associated data, it clears the temporary table and commits the transaction to ensure data integrity. This approach prevents database overload when removing large amounts of user data, ensuring that performance remains stable.

- **`optimizeTable.py`** (Database Optimization) - This script improves database efficiency by running the MySQL OPTIMIZE TABLE command on the User_Tests, Life_style and Users. The optimization process helps remove table fragmentation, reclaim unused space, and improve query performance. By keeping the table optimized, it ensures that retrieving data remains fast even as the database grows.

### 3. Data Insertion and Synthetic Data Generation Files

These files handle the process of importing user health data from CSV datasets, cleaning and formatting the data, and inserting or updating records in the database.

- **`insert_to_DB.py`** (Insert Data from CSV files) - This script handles the first data import from a CSV file into the database. It processes missing values, converting smoking (SMK_STAT) and drinking (DRK_YN) statuses into numeric values (0/1). Since lifestyle attributes are not fully available in the dataset, the script generates random values for marital status, education level, physical activity, employment, dietary habits, and sleep patterns to complete the dataset.
  Before inserting new users, the script checks if they already exist in the Users table. If a record is found, it updates the existing entry instead of creating duplicates. Lifestyle details are stored in the Life_Style table, and test results for that year are inserted into the User_Tests table. Transactions are committed in batches to keep the database efficient and minimize the risk of data corruption.

- **`update_DB.py`** (Updating Existing Data) - This script updates existing records rather than adding new ones. Unlike insert_to_DB.py, it does not generate lifestyle attributes or passwords, as these remain unchanged. Instead, it updates user ages based on predefined age groups and adds new test results for each year.
  To handle large datasets efficiently, the script processes records in chunks of 10,000 rows, reducing memory load. Before making updates, it verifies if a user already exists. If so, it updates the age, height, and weight fields while ensuring new test results are added to the User_Tests table without overwriting previous data. Using batch processing (executemany()), the script updates thousands of records quickly and prevents redundant updates.
  This process keeps user records accurate over time, allowing for reliable long-term health tracking.

## Client-Side Code (`/my_health_tracker/` folder)

The client-side of the application is built using **React** to create a dynamic and responsive user interface. We used **Bootstrap** for styling and layout design, ensuring a clean and user-friendly experience. The project is primarily developed using **JavaScript** for functionality and **CSS** for styling to maintain a consistent and visually appealing design.

**Key Files:**

- **`App.js`** – The root file of the application that manages routing and initializes the main components. It serves as the entry point and controls the navigation between different pages.
- **`LoginPage.js`** (located in the **`/loginPage/`** folder) – This file handles user authentication by displaying the login form and validating user credentials.
- **`SignUp.js`** (located in the **`/signUp/`** folder) – This file manages the registration process, allowing new users to create an account by entering their details.
- **`MainPage.js`** – The core file of the application, responsible for displaying the main dashboard after successful authentication. It integrates multiple features, including test result tracking, health risk predictions, and comparisons with similar users.

These components work together to deliver a seamless experience, connecting to the backend for data retrieval and processing while maintaining an interactive UI.

# Part 5: Machine Learning Models

In the MyHealthTracker application, we implemented **machine learning models** to provide personalized health risk predictions based on user medical and lifestyle data. Our system utilizes multiple classification algorithms, selecting the best-performing models for predicting heart disease, stroke, diabetes, and depression. The models were trained on structured medical datasets and optimized to align with the database scheme used in the application. Each prediction model takes relevant user features, applies preprocessing, and outputs a risk assessment with a probability score. These models are integrated into the backend via API endpoints, allowing real-time predictions for users.

All model files can be found in the **`'models/'`** folder.

## Models Summary

1. **Heart Disease Prediction Model**

   The Heart Disease Prediction Model was designed to assess a user's likelihood of developing cardiovascular conditions based on key health indicators.

   Features:

   - **Age** – Patient's age in years.
   - **Gender** – Encoded as `1` (Male) or `2` (Female).

- **Blood Pressure (BP)** – Categorized as normal (≤120/80 mmHg) or high (>120/80 mmHg).
- **Cholesterol Levels** – Mapped into risk categories, with high risk if total cholesterol exceeds 200 mg/dL.
- **Fasting Blood Sugar (FBS)** – A binary indicator where 1 represents glucose levels above 126 mg/dL.

Best Performing Model: **Gradient Boosting** provided the most accurate predictions by effectively structuring risk factors and improving classification accuracy.

## 2. Stroke Prediction Model

The Stroke Prediction Model estimates the probability of a user suffering from a stroke based on health and lifestyle factors.

Features:

- **Age** – Patient's age in years.
- **Gender** – Encoded as 1 (Male), 2 (Female), or 3 (Other).
- **Hypertension** – A binary indicator where 1 indicates high blood pressure (systolic BP >140 mmHg or diastolic BP >90 mmHg).
- **Smoking Status** – Categorized as:
  - 0: Never smoked
  - 1: Formerly smoked
  - 2: Currently smoking
- **Average Glucose Level** – A normalized continuous variable.
- **BMI** – Body Mass Index, a key metric for stroke risk.

Best Performing Model: The **XGBoost** model provided the most consistent and accurate predictions, especially due to its ability to handle missing data and feature variations.

## 3. Diabetes Prediction Model

The Diabetes Prediction Model determines a user's likelihood of having diabetes or pre-diabetes based on lifestyle and medical factors.

Features:

- **High Blood Pressure** – A binary indicator (1 if BP >140/90 mmHg).
- **High Cholesterol** – A binary indicator (1 if total cholesterol >200 mg/dL).
- **BMI** – Continuous variable representing body mass index.

- **Smoking Status** – A binary indicator (1 if the user is currently smoking).
- **Physical Activity** – Categorized as low (0), moderate (1), or high (2).
- **Alcohol Consumption** – A binary indicator (1 if the user is a heavy drinker).
- **Gender** – Mapped based on MyHealthTracker's schema.
- **Age Group** – Mapped into predefined database categories.
- **Education Level** – Coded from high school (1) to PhD (5).

Best Performing Model: The **XGBoost** model was chosen due to its strong adaptability to structured data and its ability to make actionable predictions.

4. **Depression Prediction Model**

The Depression Prediction Model evaluates the likelihood of mental health conditions based on lifestyle, demographic, and behavioral factors.

Features:

- **Age** – Patient's age in years.
- **Marital Status** – Mapped into predefined categories.
- **Education Level** – Mapped to database categories.
- **Number of Children** – Numeric count of dependents.
- **Smoking Status** – Coded as non-smoker (0), former smoker (1), or current smoker (2).
- **Physical Activity** – Coded as low (0), moderate (1), or high (2).
- **Work Status** – Binary indicator (1 if employed).
- **Alcohol Consumption** – Binary indicator (1 if heavy drinker).
- **Dietary Habits** – Categorized as healthy (0), moderate (1), or unhealthy (2).
- **Sleep Patterns** – Mapped into severity-based categories.

Best Performing Model: The **Neural Network** was chosen for its ability to handle complex non-linear relationships within the dataset.

## API Integration for Real-Time Predictions

The MyHealthTracker application integrates these models into its backend **Flask API**, ensuring users receive real-time risk assessments.

**Key Endpoints:**

1. **/api/predict_heart_disease** - Uses the Gradient Boosting model to analyze heart disease risk based on user data, and returns risk_level (low, moderate, or high) and probability.

2. **/api/predict_stroke** - Uses the **XGBoost model** to assess stroke risk, and returns risk_level (low or high) and probability.
3. **/api/predict_diabetes** - Applies the **XGBoost model** to classify diabetes risk, and returns risk_level (low, moderate, or high) and probability.
4. **/api/predict_depression** - Uses the **Neural Network** to evaluate depression likelihood and returns risk_level (low or high) and probability.

These endpoints interact with the MySQL database, retrieving user information, lifestyle attributes, and test results, and then applying the trained models for accurate predictions.

## Prediction Models Directory Structure

The `'/models/'` directory contains the machine learning models and scripts used for training and evaluating predictions.

**Trained Model Files (.pkl)**:

- `best_heart_disease_model.pkl`
- `best_stroke_model_Gradient Boosting.pkl`
- `best_diabetes_model_Gradient Boosting.pkl`
- `best_logistic_regression_model_depression.pkl`

**Python Scripts (.py)**:

- `heart_disease.py` – Model training for heart disease.
- `stroke.py` – Handles stroke prediction model.
- `diabetes.py` – Code for diabetes prediction.
- `depression.py` – Training script for depression model.

**Performance Graphs (.png)**:

- `model_performance_plot_heart.png` – Performance visualization for heart disease models.
- `model_performance_plot_stroke.png` – Evaluation metrics for stroke prediction models.
- `model_performance_plot_diabetes.png` – Graphical representation of diabetes model performance.
- `model_performance_plot_depression.png` – Performance comparison of depression prediction models.