Jacob Flores Gomez
12/13/2024
Documentation for my Workout Planner web application

Section 1: Introduction

Purpose

My documentation is a guide for developers taking over my Workout Planner web application. My main goal is to provide a clear understanding of the functionality of my application, while explaining the structure, logic, design and coding decisions. In addition, I highlight areas that can be improved or implemented. By following my documentation, you will save time and avoid unnecessary trial and error while exploring Workout Planner code base.

Project Overview

The My Workout Planner web application is designed to help users create, track and manage their exercise routines. It is intended for anyone who wants to use exercise to lead a healthy lifestyle and for those who want a simple tool to plan and organize their workouts. The ability for users to receive a training program based on their experience, goals and weekly availability is one of the main advantages of the application. Demonstration videos will be provided to both experienced and inexperienced users to give them a better understanding of how to perform each exercise. In addition, it features progress tracking, which presents a line graph of each exercise over time and data of all workouts performed in a dashboard. Users can easily edit or delete workouts as their fitness goals change, and a secure login system ensures that data remains private and saved.

The application is built with HTML, CSS (Bulma) and some JavaScript for the frontend. While Flask, SQL, and JSON take care of the backend. In addition to complement our app, we have implemented Llama. Llama AI is a large language model that is trained by Meta AI that helps to understand and respond to human input and develop human-like text.

- Flask handles the logic, authentication, and routing.
- Data and user information is stored in SQL.
- JSON is used to transfer and store the user's weekly schedule between the frontend and the backend.
- Llama is responsible for processing and analyzing the user's responses to provide them with a schedule and training program based on that.

Section 2: User Perspective

User Features Overview

The first thing the user will encounter when visiting the website is the landing page. It consists of three sections: home page, about us, and features. The “home page” welcomes the user by providing insights and inviting them to register. The “about us” section mentions the mission,

objectives, values, and introduces the team. While the last section “features” gives information about what the application offers to the user as mentioned above in the project overview.

After having visited the landing page, new users can create an account through the “Sign Up” button by entering their username, password, and e-mail. To protect user data, passwords are securely hashed and stored in the database. Users can access to all features by logging in after registering. Secure user sessions are handled by Flask.

After logging in, the user must complete a series of three questionnaires. The forms are easy to use because they are straightforward and sequential. The first asks about the user's training experience, fitness goals, and availability. The second form allows the user to enter their weekly schedule, i.e., classes, work, hobbies, etc. And the last questionnaire asks about the user's weight, height and body type. Below are the three questionnaires:

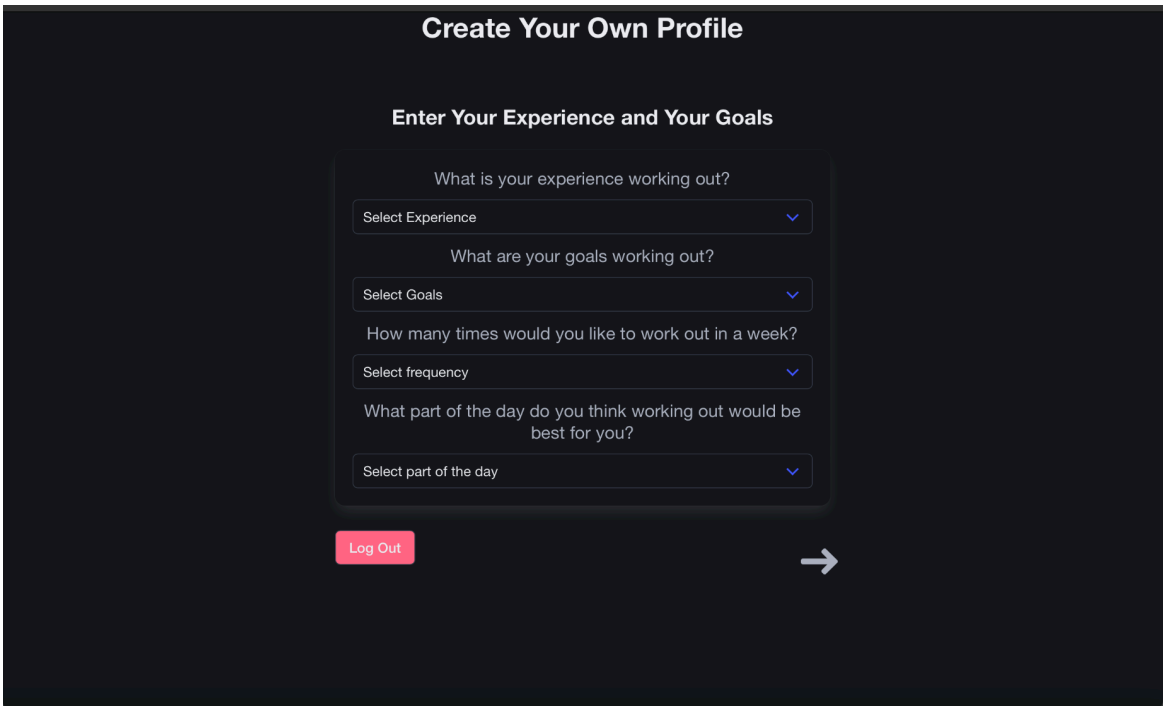

Figure 1: *User experience form*

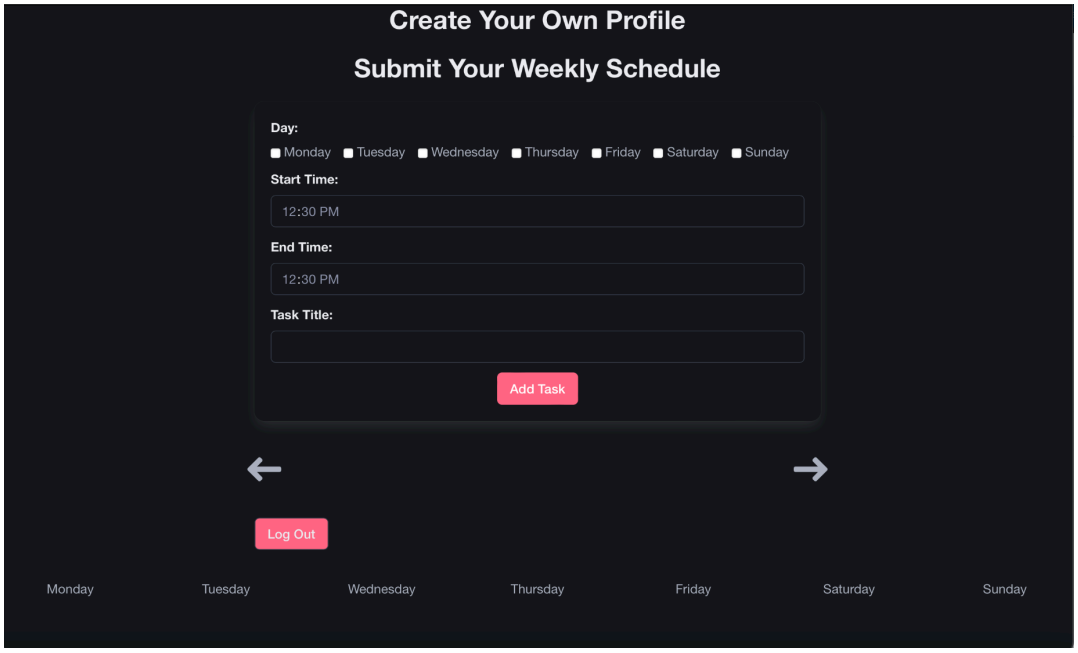

Figure 2: *Form to submit schedule*

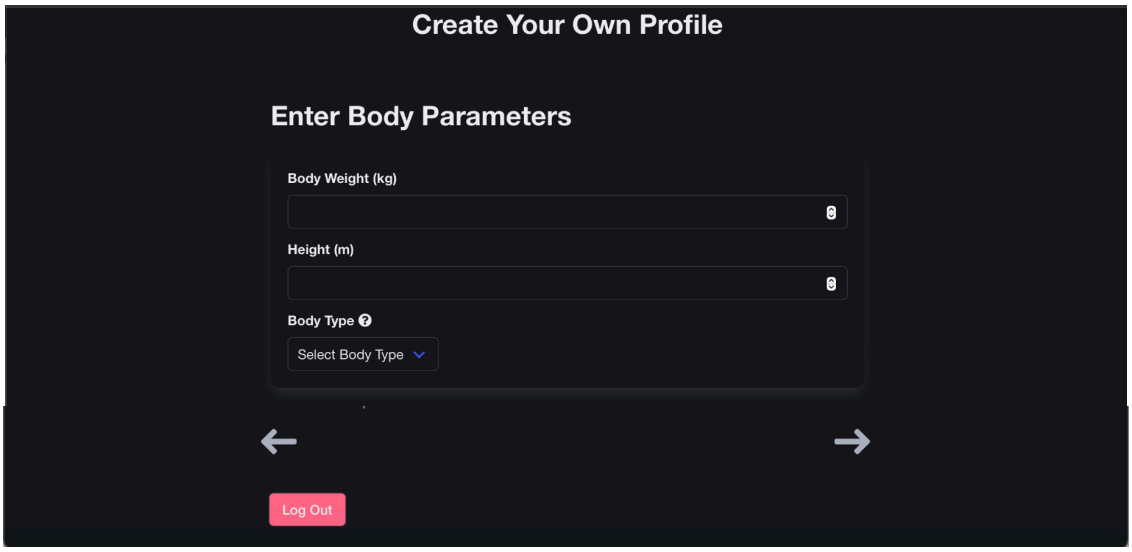

Figure 3: *Body Parameters form*

When completing the questionnaires you will have the chance to review all your responses and go back in case you need to edit your answers or you can simply go to the home page and get to know the features of the website. All this gathered user information will be used by Llama, as mentioned before, so that this machine learning tool can create a training program based on the user's answers/preferences. Llama makes sure that the user can train on days that do not overlap with other responsibilities during the week while exercising according to their needs. The user's

answers are verified and stored in the database (SQL) once submitted and displayed in the user's profile with the option to edit it at any time.

After logging in and completing the questionnaires, the user will see a navigation bar with the features: workout, schedule, progress, and profile. The workouts feature acts as the primary page after logging in. It shows the workout of the day, providing specifics such as exercises, weights, sets, reps, and even video demonstrations per exercise as mentioned above. So, it provides quick access to workout information and simplifies routine management. Also users can edit or delete their workouts.

The schedule section shows a weekly schedule, which includes the information already recorded, such as the user's responsibilities, along with the added workouts. This section also has a button to add or delete tasks, as it is possible that the user's schedule may change and not be fixed forever.

In the progress section when users log their completed workouts, the application updates this section with visual graphs and stores the information in the database to show the user their history of weights used on the correct date for each exercise. These graphs encourage users to stay consistent and motivated by making it easy to track progress trends thanks to the pandas, seaborn and matplot libraries.

The profile displays the user's information, along with the user name and e-mail address. It also includes two buttons, one to retake the questionnaires and one to change the password.

Design Decision
Workout Planner is designed with a clean and basic layout to make it easy and intuitive to use. The questionnaires were separated into three parts to make the site more interactive and less static and boring. The arrows used between the questionnaries to move forward or backward add a touch of creativity to the app. We made sure that the navigation is usable even by users unfamiliar with web applications. We used frameworks such as Bulma to ensure consistency and responsive styling, components that improve the user experience.

The design is accessible to anyone. We took into account that the main target audience is ordinary people and not technology experts, which made us focus more on functionality than complexity.

Section 3: Code Documentation

Overall Architecture

The architecture of Workout Planner is based on Flask, which connects different components and organizes them to have an ordered structure. The main base is “app.py”, which is the main script of the application. This file defines routes that are responsible for managing user interactions. It also integrates the application logic with the database and generates dynamic HTML pages using Jinja2 templates.

The “config.py” file provides classes and helper functions to support the application's schedule and time logic. It includes a Schedule class that manages user schedules with a detailed hour-by-hour breakdown of tasks. It reuses the components and reduces the writing of so much unnecessary code.

The unit tests are in the file “app_tests.py”. This part is indispensable since it is in charge of testing each isolated feature. This helped to identify bugs in the development process and improve the code quality.

The database schema, “schema.sql”, manage all the data generated from the application. It includes tables such as:

- Users: Stores authentication and user information.
- Schedule: Includes the seven days of the week to organize the weekly training plans.
- Workout: Includes about 100 exercises for chest, back, arms, abs, and legs along with their respective demostration videos.

See below part of our database:

```
create table users (
 id integer primary key autoincrement,
 username text not null,
 password text not null,
 email text not null,
 experience text,
 goals integer,
 frequency integer,
 part_of_the_day integer,
 body text,
 weight float,
 height float
);


create table schedule (
 id integer primary key autoincrement,
 username text not null,
 Monday text,
 Tuesday text,
 Wednesday text,
```

```
 Thursday text,
 Friday text,
 Saturday text,
 Sunday text
);


create table workout (
 id integer primary key autoincrement,
 workout_name text not null,
 muscle text not null,
 weighted bit default 0, /*0 is false and 1 is true stores bit variable for
boolean*/
 goals integer default 0,
 experience integer not null,
 recommended_weight_kg INTEGER default NULL
);
```

On the other hand, for the frontend we used Jinja2 templates to generate HTML pages dynamically and not have unnecessary extra code. These templates have 3 layouts: “layout.html”, “layout_form_profile.html”, and “layout_main.html”. All three have a different head and footer for each part of the application, such as the landing page, questionnaires, and main page once logged in. This way we provided a coherent structure in each specific part of the application.

User Authentication and Profile Management

Functions in app.py like login_submit and sign_up_submit ensure that passwords are securely hashed using methods like check_password_hash for login or generate_password_hash for register. Both were imported from werkzeug.security, a python package that provides a library for hashing passwords that helps to reduce the risk of storing passwords in plain text. User credentials are validated based on the entries in the user table in the database.

By importing SendGrid, a cloud-based email delivery platform, we were able to implement the forget password option at login. This does the job of sending an email to the account owner, and then sending a link to the page again to change the password. In order for the user to receive the email, they are required to enter the username and email address with which they created the account. Without these requirements, it will not be possible to change the password.

The /sign_up route manages user registration by storing credentials in the database. The /login route verifies user credentials and establishes a session upon successful verification. The /profile route allows users to view or update their data like account information, body parameters and fitness goals. In addition, we implemented flash messages to guide the user to correctly create their account. Users must meet the required parameters such as having a valid email address, and password of at least 8 characters, among others.

In the frontend, the templates “sign_up.html”, “login.html” use JINJA2, like most html files. In both cases we used the header and footer from layout.html, which are the same features of the landing page. But for “user_profile.html” we used “layout_main.html”. This file provides header and footer for the website once logged in.

Here is the code for validating the email:

```
def validate_email(email):
    # Regular expression pattern for validating an email
    pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'

    # Check if the email matches the pattern
    if re.match(pattern, email):
        return True
    else:
        return False
```

The code allows an e-mail to include letters, digits and special characters, followed by an at sign, then a domain name with letters, digits and hyphens, and a top-level domain separated by a period.

Please find below the code used for registration:

```
@app.route('/sign_up_submit', methods=['POST'])
def sign_up_submit():
    username = request.form['username']
    password = request.form['password']
    password_c = request.form['password_c']
    email = request.form['email']



    db = get_db()
    cur = db.execute('select username, password, email,id from users order by id
desc')
    users = cur.fetchall()
    unique_user = db.execute('select username from users where username = ?',
[username]).fetchone()

    if not validate_email(email):
        flash('Not a valid email')
        return render_template('sign_up.html')

    if unique_user is not None:
        flash('Not a unique username')
        return render_template('sign_up.html')
    if username == '' or password == '' or email =='':
        flash("Did not fill out information")
        return render_template('sign_up.html')
```

```
    if len(password) < 8:
        flash('Password needs to be at least 8 characters.')
        return render_template('sign_up.html')

    if password != password_c:
        flash("Passwords did not match")
        return render_template('sign_up.html')

    db.execute('insert into users (username, password, email) values  (?, ?,
?)',
               [username, generate_password_hash(password),email])

    # create initial empty schedule for a day
    schedule = config.Schedule(None)
    encoded_schedule = json.dumps(schedule.schedule)

    # put the schedule into the schedule table in the db
    db.execute('insert into schedule (username, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday, Sunday) values (?, ?, ?, ?, ?, ?, ?, ?)',
               [username, encoded_schedule, encoded_schedule, encoded_schedule,
encoded_schedule, encoded_schedule, encoded_schedule, encoded_schedule])
    db.commit()

    return render_template('login.html', users = users)
```

Workout Creation and Schedule Management

As I said before, the workout and weekly schedule will be generated by the algorithm that Llama uses. But to make that possible we have to provide it with the necessary information, i.e. user responses and their weekly availability. On the backend, the Schedule class in config.py handles the logic using JSON to manage the user's schedule data, while specific routes like /init_schedule, /init_schedule_submit, and /review_responses interact with the schedule table in the database to fetch and update the data.

Below is the code from “config.py”:

```
class Schedule:
    def __init__(self, init_schedule: dict[str:list]):
        if not init_schedule:
            self.schedule = {
            # key-value pair in the dictionary
            # with the key being an hour and value being a list of tasks that
happen at that time
            '0': [None] * 60,
            '1': [None] * 60,
            '2': [None] * 60,
            '3': [None] * 60,
            '4': [None] * 60,
```

```
            '5': [None] * 60,
            '6': [None] * 60,
            '7': [None] * 60,
            '8': [None] * 60,
            '9': [None] * 60,
            '10': [None] * 60,
            '11': [None] * 60,
            '12': [None] * 60,
            '13': [None] * 60,
            '14': [None] * 60,
            '15': [None] * 60,
            '16': [None] * 60,
            '17': [None] * 60,
            '18': [None] * 60,
            '19': [None] * 60,
            '20': [None] * 60,
            '21': [None] * 60,
            '22': [None] * 60,
            '23': [None] * 60
            }
        else:
            self.schedule = init_schedule

    def add_task(self, task: str, start_time: str, end_time: str):
        """ Adds task to the schedule. Returns the schedule with the task added
to the given time """

        start_hour = int(start_time[0:2]) # convert the start hour to int
        start_min = int(start_time[3:]) # convert the start min to int
        end_hour = int(end_time[0:2]) # convert the end hour to int
        end_min = int(end_time[3:]) # convert the end min to int
        duration_hour = end_hour - start_hour # calculate the diff in hours
        duration_min = end_min - start_min # calculate the diff in minutes
        total_duration = (duration_hour * 60) + duration_min # calculate and
convert total duration in minutes

        if start_hour == end_hour and start_min > end_min:
            return "Error: Invalid time range for the task"

        stop = False
        hour = 0
        minute = start_min
        total_added = 0

        while not stop:
            if minute == 59:
                self.schedule[str(start_hour + hour)][minute] = task
                minute = 0
                hour += 1
```

```
            else:
                self.schedule[str(start_hour + hour)][minute] = task
                minute += 1


            total_added = total_added + 1

            if total_added == total_duration:
                stop = True

    def get_all_tasks(self):
        """ Returns a list of tuples with a task title and time period of the
task """

        start_hour = 0
        start_minute = 0
        end_hour = 0
        end_minute = 0
        task = ""
        all_tasks = []

        for i in self.schedule:
            for j in range(len(self.schedule[i])):
                # checks for tasks in the given hour
                if self.schedule[i][j] is not None and task == "":
                    task = self.schedule[i][j]
                    start_hour = i # hour when the task starts
                    start_minute = j # specific minute when the task starts
                elif (task != "" and self.schedule[i][j] is None) or
(self.schedule[i][j] is not None and self.schedule[i][j] != task and task !=
""):
                    end_hour = i # assign the end hour
                    end_minute = j # assign the end minute

                    # if the digits are less than 10, put 0 before the digit
                    if int(start_hour) < 10:
                        start_hour = f"0{start_hour}"
                    if start_minute < 10:
                        start_minute = f"0{start_minute}"
                    if int(end_hour) < 10:
                        end_hour = f"0{end_hour}"
                    if end_minute < 10:
                        end_minute = f"0{end_minute}"

                    all_tasks.append((task, f"{start_hour} : {start_minute} -
{end_hour} : {end_minute}"))

                    task = ""
```

```
        return all_tasks # return statement to get all the tasks
```

This code extract contains a Schedule class that is in charge of the user's daily schedule, where each hour of the day is represented as a key from 0 to 23, and each key is assigned to a value of 60 that represent the minutes of that hour in which the tasks are performed. The add_task method allows adding tasks to specific time periods, taking the name of the task and a start and end time. This prevents tasks occurring at the same time from overlapping. On the other hand, the get_all_tasks method retrieves all tasks from the schedule as a list of tuples (task, time) in a formatted string. This code is applied at the time the user enters his schedule in the questionnaire. In this format the schedule is given to Llama to process along with the user's other responses to create the ideal schedule.

The route /init_schedule_submit, processes the user's weekly availability entry in the questionnaire. And the schedule.html template creates an interactive interface for viewing and editing training schedules. Here is the code:

```
@app.route('/init_schedule_submit', methods=['POST'])
def init_schedule_submit():
    user_id = session['user_id']
    days = request.form.getlist('day') # get the selected days
    start_time = request.form.get('start_time')
    end_time = request.form.get('end_time')
    task = request.form.get('task')

    db = get_db()
    for day in days:
        schedule_encoded = db.execute(f'select {day} from schedule where id = ?
', [user_id]).fetchone()[0] # get the schedule for the selected day
        schedule_decoded = json.loads(schedule_encoded) # decode the data
structure using json.loads() function
        new_schedule = config.Schedule(schedule_decoded) # create new object of
the Schedule class and pass the decoded schedule there
        new_schedule.add_task(task, start_time, end_time) # call the method to
add the task to the schedule
        encode_new_schedule = json.dumps(new_schedule.schedule) # encode new
schedule using json.dumps() function
        db.execute(f"update schedule set {day} = ? where id = ?",
[encode_new_schedule, user_id]) # update the schedule table in the db

    db.commit()

    return redirect(url_for('init_schedule'))
```

Workout Creation

As I mentioned, we inserted 100 exercises in the database, which include these muscle groups: chest, back, abs, arms, and legs. Llama is in charge of accessing this data and recommending the

user the ideal workout based on their experience, fitness goal, height, weight, and body type. Below you can see an example of how we have organized 20 chest exercises:

```
insert into workout (workout_name, muscle, weighted, goals, experience,
recommended_weight_kg, video) values
('Decline Barbell Bench', 'Chest', 1, 1, 1, 60,
('static/videos/chest/decline_barbell_bench')),
('Flat Barbell Bench', 'Chest', 1, 1, 2, 55,
('static/videos/chest/flat_barbell_bench')),
('Incline Dumbbell Bench', 'Chest', 1, 1, 2, 30,
('static/videos/chest/incline_dumbbell_bench')),
('Decline Dumbbell Bench', 'Chest', 1, 1, 1, 40,
('static/videos/chest/decline_dumbbell_bench')),
('Band Pushup', 'Chest', 0, 1, 2, NULL, ('static/videos/chest/band_pushup')),
('Machine Pec Fly', 'Chest', 1, 1, 2, 25,
('static/videos/chest/machine_pec_fly')),
('Diamond Pushup', 'Chest', 0, 1, 1, NULL,
('static/videos/chest/diamond_pushup')),
('Band Flat Bench', 'Chest', 0, 1, 2, NULL,
('static/videos/chest/band_flat_bench')),
('Incline Smith Machine', 'Chest', 1, 1, 1, 35,
('static/videos/chest/incline_smith_machine')),
('Chest Dip', 'Chest', 0, 1, 1, NULL, ('static/videos/chest/chest_dip')),
('Band Chest Flys', 'Chest', 0, 1, 1, NULL,
('static/videos/chest/band_chest_flys')),
('Decline Smith Machine', 'Chest', 1, 1, 3, 45,
('static/videos/chest/decline_smith_machine')),
('Cable Crossover', 'Chest', 1, 1, 1, 20,
('static/videos/chest/cable_crossover')),
('Low Chest Flys', 'Chest', 1, 1, 2, 15,
('static/videos/chest/low_chest_flys')),
('Flat Smith Machine Bench', 'Chest', 1, 1, 45, 60,
('static/videos/chest/flat_smith_machine_bench')),
('Machine Chest Press', 'Chest', 1, 2, 3, 35,
('static/videos/chest/machine_chest_press')),
('Dumbbell Chest Flys', 'Chest', 1, 1, 1, 15,
('static/videos/chest/dumbbell_chest_flys')),
('Incline Machine Press', 'Chest', 1, 1, 1, 40,
('static/videos/chest/incline_machine_press')),
('Incline Barbell Bench', 'Chest', 1, 1, 0, 30,
('static/videos/chest/incline_barbell_bench')),
('Pushups', 'Chest', 0, 1, 2, NULL, ('static/videos/chest/pushup'));
```

As you could see we organized it by name of the exercise, muscle being worked, whether it is an exercise that requires weight or not, goals, experience, recommended weight (in case weight is used), and its respective video. There are 7 columns in the workout table, where 4 contains integers. Here is the description of each of them:

- The third column is represented with a 0 or 1. 0 if the exercise does not require weight, or 1 if it does.
- The fourth column represents from 1 to 3 the goals. 1 being gain weight/bulk, 2 being loss weight, and 3 being recovery.
- The fifth column represents from 1 to 3 the fitness experience of the user. 1 being inexperienced, 2 being intermediate, and 3 being highly experienced.
- The sixth column simply recommends a general weight in kilos, by which we consider that any user regardless of his virtues could start.

The videos are stored in the static folder of the project, and we use their path in the corresponding exercise of the workout table. This way we can identify the video that belongs to each exercise and show it to the user at any time they need it.

In this way we collect all the necessary information so that Llama with the help of its algorithm can generate a complete training program based only on the parameters of the database.

Tracking progress

The progress tracking feature informs users of the weights they have used on each exercise to see whether or not they are progressing in that exercise for any muscle group. In this way they can monitor their fitness through clear linear visualizations.

The following code extends the workout table. We added a time_minutes column, which records the duration of a workout in minutes. Next, we added the user_weight_kg column to record the user's weight in kilograms. The is_current column, is added to indicate whether a workout is currently active with a default value of 0 (inactive). Next, an update query sets the is_current value to 1 (active) for rows where the id column matches 1, 2, or 3, marking those rows as current workouts. Finally, user_id column is added to associate workouts to specific users. This allows the user to insert any of these parameters and have them saved and review their progress over time. See the code below:

```
ALTER TABLE workout ADD COLUMN time_minutes INTEGER DEFAULT 0;

ALTER TABLE workout ADD COLUMN user_weight_kg INTEGER DEFAULT NULL;

ALTER TABLE workout ADD COLUMN is_current BOOLEAN DEFAULT 0;

UPDATE workout SET is_current = 1 WHERE id IN (1, 2, 3);

ALTER TABLE workout ADD COLUMN user_id INTEGER;
```

In the backend, the data is gathered from the database and fetch all workout data into a dataframe, we get the user's exercises from their workout data and put all that information together with the dates they performed them and display it in a linear graph. Libraries like Pandas and Matplotlib are used to process and analyze this data.

Below is the code we used to gather the data needed for progress section along with the code for visualizations:

```
@app.route('/progress', methods=['GET'])
def progress():
    db = get_db()
    if session['user_id']:
        selected_workout = request.args.get('workout', '').lower()
        print(selected_workout)
        if selected_workout:
            # Fetch workout details for the selected muscle group
            query = '''
                    SELECT workout_name, time_minutes, user_weight_kg
                    FROM workout
                    WHERE LOWER(muscle) = LOWER(?)
                    ORDER BY id DESC
                '''
            cur = db.execute(query, [selected_workout])
        else:
            # Fetch all workout details grouped by muscle
            query = '''
                    SELECT muscle, workout_name, time_minutes, user_weight_kg
                    FROM workout
                    ORDER BY muscle, id DESC
```

```
                '''

            cur = db.execute(query)

        exercises = cur.fetchall()

        # load workout data into a Pandas dataframe

        df = pd.read_sql("select * from workout", db)

        # fetch the unique muscle groups from the data

        muscle_groups = df['muscle'].unique()

        # initialize a list to store plot image paths

        plots = []

        # determine which muscle groups to plot based on the selected workout

        if selected_workout == "all" or not selected_workout:

            muscle_groups_to_plot = muscle_groups

        else:

            muscle_groups_to_plot = [selected_workout]

        # generate line plots for the selected muscle groups

        for muscle_group in muscle_groups_to_plot:

            # filter the dataframe for the current muscle group

            muscle_group_df = df[df['muscle'].str.lower() ==
muscle_group.lower()]

            # generate a line plot for each exercise in the muscle group

            exercises_in_group = muscle_group_df['workout_name'].unique()

            for exercise in exercises_in_group:

                exercise_df = muscle_group_df[muscle_group_df['workout_name'] ==
exercise]

                plt.figure(figsize=(14, 8))

                sns.lineplot(x='time_minutes', y='user_weight_kg',
data=exercise_df, marker='o')
```

```
                plt.title(f'Weight progression for {exercise}
({muscle_group.capitalize()})')

                plt.xlabel('Time')

                plt.ylabel('Weight (kg)')

                plt.xticks(rotation=45)

                plt.grid(axis='y')

                # save the plot image

                plot_filename =
f'static/plots/{muscle_group.lower()}_{exercise.lower().replace(" ",
"_")}_progression.png'

                plt.savefig(plot_filename)

                plt.close()

                # add the plot file path to the list

                plots.append(plot_filename)

        return render_template('progress_layout.html',
selected_workout=selected_workout,exercises=exercises, plots=plots)
```

Key Design Decisions

We chose SQL as the database because of its simplicity and good integration with Flask. The database schema was designed to store all user information that can contribute to the purpose of the application, and to allow future extensions with JSON.

On the frontend, we used Jinja2 templates to be able to render, minimizing code duplication. The “layout_main.html” template ensures a consistent header and footer in the application once logged in. The styling is handled with Bulma, as mentioned before. We also used some JavaScript to made the arrows work between questionnaries.

Section 4: Features Yet to Be Implemented

This section describes the features that could be implemented in a future, along with an overview, tools needed, integration techniques, and possible challenges.

Workout sharing

A good idea would be a workout sharing feature, which allows users to share their workout plans or stats with friends. I think it would be interesting if users could invite others through links, so that we can reach more people. Also, shared workouts would appear as read-only with an option to suggest changes. I think secure sharing would be based on token access, while REST and JSON APIs would take care of the data exchange. On the other hand, I don't think we need a new table, as we already have enough information with having the owner id and recipient id. Developers would need new paths, such as profile/share_friend, and a share button in the "schedule.html" template for easy access. Challenges could be ensuring complete privacy, avoiding data leakage, and managing expired links.

Google Calendar

Another interesting idea would be to add third-party APIs such as Google Calendar. Users could import activity data such as workouts, appointments and events, which would further enhance the training suggestions provided by the app. Python libraries would handle communication with the Google Calendar API. Developers would need experience using Google Calendar API and data formats. To implement this to our current code, a new path such as /connect_calendar would need to be created for authentication. While the /schedule path would be updated to merge the imported data with the existing records. Challenges would include ensuring reliable data synchronization and seamlessly integrating external and application data without inconsistencies.