ReadMe.md

```markdown
# Health and Fitness Tracking App

## Overview

The Health and Fitness Tracking App is a digital platform designed to assist users in monitoring and understanding their health and fitness data. The app aims to empower individuals to make informed decisions about their lifestyle and wellness routines by leveraging a robust database and user-friendly interface.

## Objectives

- **Personalized Health Monitoring:** Enable users to track specific health metrics tailored to their needs.
- **Workout Logging:** Users can log various workout details, including type, duration, and intensity.
- **Nutritional Tracking:** The app comprehensively monitors users' nutritional intake, focusing on calorie tracking and macro breakdowns.
- **Sleep Pattern Analysis:** Users can record and analyze their sleep duration and quality to better understand their rest cycles.
- **Mental Wellness:** Track mental health activities, stress levels, and general mood to support overall well-being.
- **Health Recommendations:** Provide personalized health recommendations to users based on their logged data.

## Target Audience

This app's primary audience includes health-conscious individuals looking to improve or maintain their fitness and well-being through data-driven insights.

## Data Stored

- `users`: Stores personal and authentication details of each user along with links to their health and fitness records.
- `body_measurements`: Keeps track of user body metrics like height and weight over time.
- `workout_types`: Defines different types of workouts that users can log.
- `workouts`: Records details of each workout session, including type, duration, intensity, and calories burned.
- `meal_types`: Categorizes types of meals, such as breakfast, lunch, dinner, etc.
- `nutrition`: Logs nutritional intake details, including calorie count and macronutrient breakdown for user meals.
```

- `sleep`: Captures data on user sleep patterns, including duration and quality of sleep.
- `health_metrics`: Tracks various health metrics such as heart rate and blood pressure for users.
- `mood_types`: Lists types of moods that users can record for mental health tracking.
- `mental_health`: Logs mental health data, including stress levels, mood types, and mindfulness activities.
- `health_recommendations`: Contains personalized health and fitness recommendations for each user.

Database Schema Design

The database schema provides a comprehensive overview of users' health and fitness data while ensuring data integrity and efficient querying. Primary keys ensure the uniqueness of records, while foreign keys enforce relationships between different data entities. Indexes on frequently searched fields like `username,` `email,` and `recorded_date` across various tables optimize query performance. Data validation is also applied to ensure data input is within reasonable expected range.

Database Design Considerations

Primary and Foreign Key Usage

- **Primary Keys** are unique identifiers for each record within a table. In this database, they ensure that each user, workout, nutrition log, etc., has a unique identifier, such as `id,` which is crucial for indexing and managing relationships between tables.

- **Foreign Keys** establish a link between related data across different tables. For example, `user_id` in the `workouts` table references the `id` in the `users` table, enabling the database to maintain consistent and coherent data about which workouts belong to which users.

Normalization

Normalization is organizing data in a database to reduce redundancy and improve data integrity. This database is normalized to the Third Normal Form (3NF), building upon the requirements of the First (1NF) and Second (2NF) Normal Forms.

First Normal Form (1NF)

```markdown
- **Atomicity**: Each field contains indivisible atomic values with no repeating groups or arrays.
- **Unique Identifiers**: A primary key in each table ensures each record is unique and identifiable.
- **Data Consistency**: Using consistent data types and formats across similar fields prevents data conflicts.

By satisfying 1NF, the database ensures a solid foundation where each table represents a single entity or concept, and each row/column intersection contains a single value.

#### Second Normal Form (2NF)

- **Elimination of Partial Dependencies**: By ensuring that all non-key attributes are fully functionally dependent on the primary key, not just part of it, the database eliminates partial dependencies.
 This step requires the presence of a primary key, and in the case of composite primary keys, it ensures that no attribute is dependent on only a part of the key. Thus, all the fields in a table relate directly to the primary key, enhancing data retrieval speed and consistency.

#### Third Normal Form (3NF)

- **Removal of Transitive Dependencies**: Non-key attributes are not allowed to depend on other non-key attributes, which means all attributes are directly dependent on the primary key.
 By reaching 3NF, the database avoids transitive dependencies, which not only keeps data redundancy to a minimum but also protects data integrity by preventing anomalies that can occur during data operations (inserts, updates, or deletions).

### Importance of 3NF

Maintaining 3NF is crucial for several reasons:

- **Data Integrity**: There's a single source of truth for each piece of information, which prevents data duplication and inconsistency.
- **Maintenance**: Simplifies maintenance tasks because updates, inserts, or deletes need to happen in only one place.
- **Performance**: Improves query performance due to reduced data duplication and a more efficient structure.
- **Flexibility**: Makes the database more adaptable to changes, as adjustments for evolving business requirements can be made without extensive redesign.
```

- **Clarity**: Provides more apparent relationships between entities, which can be crucial for developers and analysts when understanding the database structure.

The careful design aligned with 1NF, 2NF, and 3NF principles is essential for a robust, reliable, and efficient database system like the Health and Fitness Tracking App.

Query Optimization with Indices
Indices in this health and fitness tracking application are used to speed up data retrieval from the database. They are especially valuable when querying large datasets, as they prevent the need for full table scans. For instance, the `users` table has indices on `username` and `email,` which are likely to be used often to look up user information, thus accelerating search operations. Similarly, tables like `body_measurements,` `workouts,` `nutrition,` `sleep,` `health_metrics,` and `mental_health` have indices on user-related foreign keys and dates. Queries often filter or sort based on dates and user IDs. Indices on these columns mean the database can quickly locate and retrieve the relevant records without scanning the table. This leads to faster response times for end-users and reduced load on the database server, which is critical for providing a smooth user experience.

Transactions and ACID Concepts in Data Insertion

The insert_data file illustrates database transactions, a key part of the ACID properties (Atomicity, Consistency, Isolation, Durability) essential for maintaining database integrity.

Atomicity
Atomicity is demonstrated through transactions when adding new users and related records like body measurements and health metrics. The code ensures that all operations within a transaction block are treated as a single unit, meaning that they either succeed or are not applied. This is achieved by wrapping user creation and related data insertions within a `try` block, flushing to obtain user IDs for related records without committing, and then committing all at once if all insertions are successful.

Consistency
Consistency is maintained by enforcing data integrity constraints and relationships. If a transaction fails, a `rollback` is initiated to undo any changes made during the transaction, ensuring the database remains consistent.

Isolation

```
Isolation is implicitly managed by SQLAlchemy's session and transaction management
system. Each transaction is isolated from others, ensuring that the operations within
a transaction are completed without interference from other concurrent transactions.

### Durability
Durability is guaranteed by committing the transactions to the database. Once the
transaction is executed, the changes are permanent, even in the event of a system
failure, ensuring the reliability of the data.

The use of transactions in this code shows a commitment to the ACID properties,
ensuring that the database operations are reliable, consistent, and correct.

## Database Query Intentions

The SQL queries in the code aim to extract specific insights from a health and fitness
tracking application's database:

### Total Calories Burned Per User
This query calculates the sum of calories each user burns through their workout
sessions, giving an overview of their total energy expenditure.

### Latest Body Measurements Per User
This query retrieves each user's most recent height and weight measurements, providing
up-to-date information on their physical dimensions.

### Average Sleep Duration
This query determines the average sleep duration for each user over the past month,
offering insight into their sleep patterns.

### Top 5 Frequent Workout Types
This query identifies the five most common workout types across all users,
highlighting the most popular fitness activities within the app's community.

### Last Nutrition Log Entry
This query finds the details of each user's last recorded nutrition log, showing their
most recent dietary intake.

### Average Heart Rate Per User
This query computes the average heart rate for each user, which can indicate
cardiovascular fitness or stress.

### Latest Mental Health Log
```

~~~
This query fetches each user's latest recorded mental health log, revealing their most recent stress levels and mindfulness activity durations.

## File Overview

### `create.py`
- **Purpose**: Establishes the structure of the database.
- **Functionality**: Contains class definitions for each table using SQLAlchemy ORM, sets up the database engine, and executes commands to create the database with the defined schema.

### `insert_data.py`
- **Purpose**: Populates the database with data.
- **Functionality**: Utilizes the Faker library to generate and insert randomized but realistic data into the tables for users, body measurements, workouts, meal types, nutrition logs, sleep records, health metrics, and mental health logs.

### `query_data.py`
- **Purpose**: Retrieves and displays information from the database.
- **Functionality**: Contains various SQL queries executed via SQLAlchemy ORM that pull and aggregate data from the database, such as total calories burned per user, latest body measurements, average sleep duration, frequent workout types, last nutrition log, and latest mental health logs. Outputs the results of these queries to the console.

## Setting Up the Development Environment

Instructions for setting up the Python virtual environment and installing dependencies.

```bash
python3 -m venv venv
source venv/bin/activate
pip3 install -r requirements.txt

python create.py
python insert_data.py
python query_data.py
```

## Testing
~~~

```
DB Browser for SQLite was used for testing.
```

Create.py:

```
from sqlalchemy import (
    create_engine, Column, Integer, String, Float, DateTime, ForeignKey, Boolean, Index
)
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, sessionmaker
import datetime

engine = create_engine('sqlite:///health_system.db')

Base = declarative_base()

class User(Base):
    """
    Representation of a user in the system. Holds the personal details and
authentication information,
    along with relationships to various health and fitness records.
    """
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True, autoincrement=True)
    username = Column(String, nullable=False, unique=True)
    email = Column(String, nullable=False, unique=True)
    password_hash = Column(String, nullable=False)
    date_of_birth = Column(DateTime, nullable=False)

    # Relationships to various records that belong to the user.
    workouts = relationship("Workout", back_populates="user")
    measurements = relationship("BodyMeasurement", back_populates="user")
    nutrition_logs = relationship("Nutrition", back_populates="user")
    sleep_records = relationship("Sleep", back_populates="user")
    health_metrics = relationship("HealthMetric", back_populates="user")
    mental_health_logs = relationship("MentalHealth", back_populates="user")
    health_recommendations = relationship("HealthRecommendation",
back_populates="user")

    # Indexes for quick lookup on frequently searched fields
    __table_args__ = (
        Index('ix_users_username', 'username'),
        Index('ix_users_email', 'email'),
    )

 class BodyMeasurement(Base):
    """
    Body measurements for users, such as height and weight at different points in time.
    """
    __tablename__ = 'body_measurements'
    id = Column(Integer, primary_key=True, autoincrement=True)
    user_id = Column(Integer, ForeignKey('users.id'), nullable=False)
    height = Column(Float)
    weight = Column(Float)
    recorded_date = Column(DateTime, default=datetime.datetime.utcnow)

    # Index for efficient querying by date
    __table_args__ = (Index('ix_body_measurements_recorded_date', 'recorded_date'),)

    user = relationship("User", back_populates="measurements")

 class WorkoutType(Base):
    """
    Different types of workouts that can be logged in the system.
    """
    __tablename__ = 'workout_types'
    id = Column(Integer, primary_key=True, autoincrement=True)
    type = Column(String, nullable=False, unique=True)

    # Relationship with Workout - one workout type can be associated with many workouts
    workouts = relationship("Workout", back_populates="workout_type")

 class Workout(Base):
    """
    Workout sessions logged by users. Stores details like duration, intensity, and
 calories burned.
    """
    __tablename__ = 'workouts'
    id = Column(Integer, primary_key=True, autoincrement=True)
    user_id = Column(Integer, ForeignKey('users.id'), nullable=False)
    workout_type_id = Column(Integer, ForeignKey('workout_types.id'), nullable=False)
    duration = Column(Integer, nullable=False)
    intensity_level = Column(Integer)
    calories_burned = Column(Integer, nullable=False)
```

```python
    date = Column(DateTime, default=datetime.datetime.utcnow)

    # Index for efficient querying by date
    __table_args__ = (
        Index('ix_workouts_user_id_date', 'user_id', 'date'),)

    user = relationship("User", back_populates="workouts")
    workout_type = relationship("WorkoutType", back_populates="workouts")

class MealType(Base):
    """
    Types of meals that can be recorded (e.g., breakfast, lunch, dinner).
    """
    __tablename__ = 'meal_types'
    id = Column(Integer, primary_key=True, autoincrement=True)
    type = Column(String, nullable=False, unique=True)

class Nutrition(Base):
    """
    Nutritional intake records, detailing calorie count and macro breakdown for meals.
    """
    __tablename__ = 'nutrition'
    id = Column(Integer, primary_key=True, autoincrement=True)
    user_id = Column(Integer, ForeignKey('users.id'), nullable=False)
    meal_type_id = Column(Integer, ForeignKey('meal_types.id'), nullable=False)
    calories = Column(Integer, nullable=False)
    protein = Column(Float)
    carbs = Column(Float)
    fats = Column(Float)
    date = Column(DateTime, default=datetime.datetime.utcnow)

    # Index for efficient querying by date
    __table_args__ = (
        Index('ix_nutrition_user_id_date', 'user_id', 'date'),)

    user = relationship("User", back_populates="nutrition_logs")

class Sleep(Base):
    """
    The Sleep class maps to the 'sleep' table and includes information about the user's
sleep patterns.
    It tracks the duration, quality, and date of sleep.
    Attributes:
        id (Integer): The primary key that uniquely identifies the sleep record.
        user_id (Integer): A foreign key that links to the 'users' table.
        duration (Float): The total number of hours slept.
        quality (Integer): An assessment of the sleep quality on a given scale.
        date (DateTime): The date and time when the sleep data was recorded.
    """

    __tablename__ = 'sleep'
    id = Column(Integer, primary_key=True, autoincrement=True)
    user_id = Column(Integer, ForeignKey('users.id'), nullable=False)
    duration = Column(Float, nullable=False)
    quality = Column(Integer)
    date = Column(DateTime, default=datetime.datetime.utcnow)

    # Index for efficient querying by date
    __table_args__ = (
        Index('ix_sleep_user_id_date', 'user_id', 'date'),)

    user = relationship("User", back_populates="sleep_records")

class HealthMetric(Base):
    """
    The HealthMetric class represents a table for tracking various health metrics.
    It stores metrics such as heart rate and blood pressure.

    Attributes:
        id (Integer): The primary key for the health metrics record.
        user_id (Integer): A foreign key that references the 'users' table.
        heart_rate (Integer): The user's heart rate in beats per minute.
        blood_pressure (String): The user's blood pressure readings.
        recorded_date (DateTime): The date and time when the metrics were recorded.
    """

    __tablename__ = 'health_metrics'
    id = Column(Integer, primary_key=True, autoincrement=True)
    user_id = Column(Integer, ForeignKey('users.id'), nullable=False)
    heart_rate = Column(Integer)
    blood_pressure = Column(String)
    recorded_date = Column(DateTime, default=datetime.datetime.utcnow)
    # Index for efficient querying by user and date
    __table_args__ = (
        Index('ix_health_metrics_user_id_recorded_date', 'user_id', 'recorded_date'),)

    user = relationship("User", back_populates="health_metrics")

class MoodType(Base):
    """
    The MoodType class maps to the 'mood_types' table and defines various types of
moods that can be recorded.

    Attributes:
        id (Integer): The primary key for the mood type record.
        mood (String): The descriptor of the mood type.
    """

    __tablename__ = 'mood_types'
    id = Column(Integer, primary_key=True, autoincrement=True)
    mood = Column(String, nullable=False, unique=True)

class MentalHealth(Base):
    """
    The MentalHealth class tracks mental health-related data including stress levels,
    mindfulness activity duration, and general descriptions of the user's mental health
activities.

    Attributes:
        id (Integer): The primary key for the mental health record.
        user_id (Integer): A foreign key that links to the 'users' table.
        mood_type_id (Integer): A foreign key that references the 'mood_types' table to
identify the mood type.
        stress_level (Integer): A numerical indicator of the user's stress level.
        mindfulness_duration (Integer): Time spent on mindfulness activities in
minutes.
        activity_description (String): A brief description of the mental health-related
activity.
        recorded_date (DateTime): The date and time the mental health data was logged.
    """

    __tablename__ = 'mental_health'
    id = Column(Integer, primary_key=True, autoincrement=True)
    user_id = Column(Integer, ForeignKey('users.id'), nullable=False)
```

```
   mood_type_id = Column(Integer, ForeignKey('mood_types.id'), nullable=False)
   stress_level = Column(Integer)
   mindfulness_duration = Column(Integer)
   activity_description = Column(String)
   recorded_date = Column(DateTime, default=datetime.datetime.utcnow)

   user = relationship("User", back_populates="mental_health_logs")

   __table_args__ = (
       Index('ix_mental_health_user_id_recorded_date', 'user_id', 'recorded_date'),)

class HealthRecommendation(Base):
   """
   The HealthRecommendation class is for storing personalized health recommendations
for users.
   Recommendations can be marked as active or inactive.

   Attributes:
       id (Integer): The primary key for the health recommendation.
       user_id (Integer): A foreign key that references the 'users' table.
       recommendation_text (String): The content of the health recommendation.
       is_active (Boolean): Flag to indicate if the recommendation is currently
active.
       created_date (DateTime): The date and time the recommendation was created.
   """

   __tablename__ = 'health_recommendations'
   id = Column(Integer, primary_key=True, autoincrement=True)
   user_id = Column(Integer, ForeignKey('users.id'), nullable=False)
   recommendation_text = Column(String, nullable=False)
   is_active = Column(Boolean, default=True)
   created_date = Column(DateTime, default=datetime.datetime.utcnow)

   user = relationship("User", back_populates="health_recommendations")

def get_new_session(database_url):
   """
   Create and return a new session for the database connection.

   Args:
       database_url (str): The URL for the database connection.
```

```
    Returns:
        Session: A SQLAlchemy session object.
    """
    engine = create_engine(database_url)
    Session = sessionmaker(bind=engine)
    return Session()

def create_tables():
    """
    Create all tables in the database according to the defined classes and
relationships.
    """
    Base.metadata.create_all(engine)

create_tables()
```

Insert_data.py

```
from faker import Faker
import random
from create import User, BodyMeasurement, WorkoutType, Workout, MealType, Nutrition,
Sleep, HealthMetric, MoodType, MentalHealth, HealthRecommendation, engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.exc import SQLAlchemyError

# Initialize Faker for data generation
fake = Faker()

# Set up database session
Session = sessionmaker(bind=engine)
session = Session()

def populate_users(session, user_count_range=(5, 20)):
    """Populate the database with random users."""
    for _ in range(random.randint(*user_count_range)):
        user = User(
            username=fake.user_name(),
            email=fake.email(),
            password_hash=fake.sha256(raw_output=False),
            date_of_birth=fake.date_of_birth()
        )
        session.add(user)
```

```python
populate_users(session)
session.commit()  # Users need to exist before related records can be added

users = session.query(User).all()

# Helper function to add random body measurements for users
def add_body_measurements(session, users, measurement_range=(1, 3)):
   for user in users:
       for _ in range(random.randint(*measurement_range)):
           measurement = BodyMeasurement(
               user_id=user.id,
               height=random.uniform(1.5, 2.0),
               weight=random.uniform(50.0, 100.0),
               recorded_date=fake.past_date()
           )
           session.add(measurement)

add_body_measurements(session, users)

# Predefined workout types to add to the database
workout_types = ['Running', 'Swimming', 'Cycling', 'Yoga', 'Weight Training']
for workout in workout_types:
   workout_type = WorkoutType(type=workout)
   session.add(workout_type)

session.commit()  # WorkoutType IDs are needed for Workout records

# Populate workouts for each user
def populate_workouts(session, users, workout_type_ids, workout_range=(5, 20)):
   for user in users:
       for _ in range(random.randint(*workout_range)):
           workout = Workout(
               user_id=user.id,
               workout_type_id=random.choice(workout_type_ids),
               duration=random.randint(20, 120),
               intensity_level=random.randint(1, 10),
               calories_burned=random.randint(100, 700),
               date=fake.past_date()
           )
           session.add(workout)

workout_type_ids = [wt.id for wt in session.query(WorkoutType).all()]
```

```python
populate_workouts(session, users, workout_type_ids)

# Add meal types
meal_types = ['Breakfast', 'Lunch', 'Dinner', 'Snack']
for meal in meal_types:
   meal_type = MealType(type=meal)
   session.add(meal_type)

session.commit()

# Generate random nutrition logs
meal_type_ids = [mt.id for mt in session.query(MealType).all()]
for user in users:
   for _ in range(random.randint(5, 20)):
       nutrition = Nutrition(
           user_id=user.id,
           meal_type_id=random.choice(meal_type_ids),
           calories=random.randint(100, 900),
           protein=random.uniform(10.0, 30.0),
           carbs=random.uniform(20.0, 50.0),
           fats=random.uniform(5.0, 20.0),
           date=fake.past_date()
       )
       session.add(nutrition)

# Generate random sleep records
for user in users:
   for _ in range(random.randint(5, 20)):
       sleep_record = Sleep(
           user_id=user.id,
           duration=random.uniform(4.0, 12.0),
           quality=random.randint(1, 10),
           date=fake.past_date()
       )
       session.add(sleep_record)

# Generate random health metrics
for user in users:
   for _ in range(random.randint(5, 20)):
       health_metric = HealthMetric(
           user_id=user.id,
           heart_rate=random.randint(60, 100),
            blood_pressure=f"{random.randint(100, 140)}/{random.randint(60, 90)}",
            recorded_date=fake.past_date()
        )
        session.add(health_metric)

# Generate mood types
moods = ['Happy', 'Sad', 'Angry', 'Excited', 'Stressed']
for mood in moods:
    mood_type = MoodType(mood=mood)
    session.add(mood_type)

session.commit()

# Generate mental health logs
mood_type_ids = [mt.id for mt in session.query(MoodType).all()]
for user in users:
    for _ in range(random.randint(5, 20)):
        mental_health_log = MentalHealth(
            user_id=user.id,
            mood_type_id=random.choice(mood_type_ids),
            stress_level=random.randint(1, 10),
            mindfulness_duration=random.randint(5, 60),
            activity_description=fake.text(max_nb_chars=200),
            recorded_date=fake.past_date()
        )
        session.add(mental_health_log)

session.commit()
session.close()  # Always close the session when done

session = Session()

try:
    # Begin a transaction to ensure atomicity of user creation and related records
    # Atomicity guarantees that a series of database operations are treated as a single
unit,
    # which either all succeed or all fail, thus preventing partial updates that could
lead
    # to data inconsistency. This is critical when operations are interdependent, such
as
    # creating a user and their associated health records.
```

```python
    # Create a new user with generated attributes
    new_user = User(
        username=fake.user_name(),
        email=fake.email(),
        password_hash=fake.sha256(raw_output=False),
        date_of_birth=fake.date_of_birth()
    )
    session.add(new_user)

    # Flush the session to get the user ID generated by the database
    # This ID is required for the foreign key relationships of subsequent records.
    # Flushing within the transaction ensures the ID is available without committing
the transaction,
    # maintaining the atomicity of the operation.
    session.flush()

    # Body measurements and health metrics are related to the user and are critical to
the user's profile.
    # By including these operations in the same transaction, we ensure that either all
user-related
    # data is persisted correctly or none at all, maintaining the referential integrity
and
    # avoiding orphan records in case of failure.

    # Add body measurements for the new user
    new_measurement = BodyMeasurement(
        user_id=new_user.id,
        height=random.uniform(1.5, 2.0),  # Meters
        weight=random.uniform(50.0, 100.0),  # Kilograms
        recorded_date=fake.past_date()
    )
    session.add(new_measurement)

    # Add health metrics for the new user
    new_health_metric = HealthMetric(
        user_id=new_user.id,
        heart_rate=random.randint(60, 100),  # BPM
        blood_pressure=f"{random.randint(100, 140)}/{random.randint(60, 90)}",
        recorded_date=fake.past_date()
    )
    session.add(new_health_metric)
    # Commit the transaction
    # Only after all related records are successfully added do we commit the
transaction,
    # ensuring the database reflects a complete set of changes.
    session.commit()

except SQLAlchemyError as e:
    # Roll back the transaction on error
    # If any operation within the transaction fails, rollback is invoked to undo all
operations,
    # ensuring that no incomplete or invalid data is left in the database, thus
preserving data integrity.
    session.rollback()
    print(f"Transaction failed: {e}")
    raise
finally:
    # Always close the session
    # Independent of the transaction's success or failure, the session is closed to
free resources.
    # This is a best practice for database connection management.
    session.close()

def populate_health_recommendations(session, users, recommendation_range=(1, 5)):
    """Populate the database with random health recommendations for users."""
    try:
        # Begin a transaction for health recommendations
        for user in users:
            # Make sure to access the user's id while the session is active
            user_id = user.id
            for _ in range(random.randint(*recommendation_range)):
                health_recommendation = HealthRecommendation(
                    user_id=user_id,
                    recommendation_text=fake.sentence(nb_words=6),
                    is_active=fake.boolean(chance_of_getting_true=75),
                    created_date=fake.past_date()
                )
                session.add(health_recommendation)

        # Commit the transaction after all health recommendations have been added
        session.commit()

    except SQLAlchemyError as e:
        # Roll back the transaction if any error occurs
        session.rollback()
        print(f"Transaction failed: {e}")
        raise

 # Ensure that the session is not closed until all operations are complete
 session = Session()

 # Pre-fetch user IDs if they are not already loaded
 users = session.query(User).all()  # This should load the user IDs

 # Call the function to populate health recommendations
 populate_health_recommendations(session, users)

 # Close the session after all operations are complete
 session.close()
```

Query_data.py

```
# Import necessary libraries and modules from SQLAlchemy and datetime package.
from sqlalchemy import func, and_
from datetime import datetime, timedelta

# Import classes for each table in the database and the engine object from create.py.
from create import (
   User, BodyMeasurement, WorkoutType, Workout, MealType,
   Nutrition, Sleep, HealthMetric, MentalHealth, engine
)

# Import sessionmaker for creating a session with the database.
from sqlalchemy.orm import sessionmaker

# Set up a new session factory bound to the engine.
Session = sessionmaker(bind=engine)
session = Session()  # Instantiate a session.

# --- Calories Burned Query ---
# Query to find the total number of calories burned per user.
# Joins User and Workout tables, groups by username, and sums the calories burned.
total_calories_per_user = session.query(
   User.username,
   func.sum(Workout.calories_burned).label('total_calories_burned')
```

```python
).join(User.workouts).group_by(User.username)

print("Total Calories Burned Per User:")
for username, total_calories in total_calories_per_user:
   print(f"{username:20} | {total_calories:10} calories")
print("-" * 35)

# --- Latest Body Measurements Query ---
# Subquery to get the latest body measurement date for each user.
latest_body_measurement_subquery = session.query(
   BodyMeasurement.user_id,
   func.max(BodyMeasurement.recorded_date).label('latest_date')
).group_by(BodyMeasurement.user_id).subquery('latest_measurements')

# Main query to get the latest body measurements using the subquery.
latest_body_measurement = session.query(
   User.username,
   BodyMeasurement.height,
   BodyMeasurement.weight
).join(User.measurements).join(
   latest_body_measurement_subquery,
   and_(
       BodyMeasurement.user_id == latest_body_measurement_subquery.c.user_id,
       BodyMeasurement.recorded_date == latest_body_measurement_subquery.c.latest_date
   )
).all()

print("Users and Their Latest Body Measurements:")
for username, height, weight in latest_body_measurement:
   print(f"{username:20} | Height: {height:10} cm | Weight: {weight:10} kg")
print("-" * 60)

# --- Average Sleep Duration Query ---
# Query to find the average sleep duration per user over the past month.
# Filters Sleep records to the last 30 days and calculates the average duration.
average_sleep_duration = session.query(
   User.username,
   func.avg(Sleep.duration).label('average_sleep_duration')
).join(User.sleep_records).filter(
   Sleep.date.between(datetime.now() - timedelta(days=30), datetime.now())
).group_by(User.username)
```

```
print("Average Sleep Duration Over the Last Month:")
for username, average_duration in average_sleep_duration:
   print(f"{username:20} | {average_duration:10.2f} hours")
print("-" * 35)

# --- Frequent Workout Types Query ---
# Query to find the top 5 most frequent workout types.
top_workout_types = session.query(
   WorkoutType.type,
   func.count(Workout.id).label('frequency')
).join(WorkoutType.workouts).group_by(WorkoutType.type).order_by(
   func.count(Workout.id).desc()
).limit(5)

print("Top 5 Most Frequent Workout Types:")
for workout_type, frequency in top_workout_types:
   print(f"{workout_type:20} | {frequency:10} times")
print("-" * 35)

# --- Last Nutrition Log Query ---
# Subquery to get the last nutrition log date for each user.
last_nutrition_date_subquery = session.query(
   Nutrition.user_id,
   func.max(Nutrition.date).label('last_date')
).group_by(Nutrition.user_id).subquery('last_nutrition_log')

# Query to get nutrition details on the last logged day by joining with the subquery.
nutrition_on_last_logged_day = session.query(
   User.username,
   Nutrition.calories,
   Nutrition.protein,
   Nutrition.carbs,
   Nutrition.fats
).join(User.nutrition_logs).join(
   last_nutrition_date_subquery,
   and_(
       Nutrition.user_id == last_nutrition_date_subquery.c.user_id,
       Nutrition.date == last_nutrition_date_subquery.c.last_date
   )
).all()

print("Users' Nutrition Intake on Their Last Logged Day:")
```

```python
for username, calories, protein, carbs, fats in nutrition_on_last_logged_day:
   print(f"{username:20} | Calories: {calories:10} kcal | Protein: {protein:5}g | 
Carbs: {carbs:5}g | Fats: {fats:5}g")
print("-" * 85)

# --- Health Metrics and Mental Health Queries ---
# Query for average heart rate per user.
average_heart_rate = session.query(
   User.username,
   func.avg(HealthMetric.heart_rate).label('average_heart_rate')
).join(User.health_metrics).group_by(User.username)

print("Average Heart Rate Per User:")
for username, avg_hr in average_heart_rate:
   print(f"{username:20} | {avg_hr:10.2f} bpm")
print("-" * 35)

# Subquery to get the latest mental health log date per user.
latest_mental_health_subquery = session.query(
   MentalHealth.user_id,
   func.max(MentalHealth.recorded_date).label('latest_date')
).group_by(MentalHealth.user_id).subquery('latest_mental_health')

# Query for the latest mental health log per user using the subquery for the latest 
date.
latest_mental_health = session.query(
   User.username,
   MentalHealth.stress_level,
   MentalHealth.mindfulness_duration
).join(User.mental_health_logs).join(
   latest_mental_health_subquery,
   and_(
       MentalHealth.user_id == latest_mental_health_subquery.c.user_id,
       MentalHealth.recorded_date == latest_mental_health_subquery.c.latest_date
   )
).all()

print("Users and Their Latest Mental Health Logs:")
for username, stress_level, mindfulness_duration in latest_mental_health:
   print(f"{username:20} | Stress Level: {stress_level:10} | Mindfulness Duration: 
{mindfulness_duration:10} min")
print("-" * 85)
```

```python
# Close the session after you're done with the queries to release resources.
session.close()
```