ChatGPT

Hospital Dashboard Backend System

Relational Database Schema

We define three main tables – **Patients**, **FollowUps**, and **Deliveries** – linked by patient IDs. The schema is normalized so that each visit or delivery is a separate record tied to a patient. For example, `patients.patient_id` is the primary key, and both `followups.patient_id` and `deliveries.patient_id` are foreign keys referencing it. Key tables and fields:

- **patients** (`patient_id` PK, `height`, `weight`, `education`, `income`, `age_at_menarche`, `age_at_marriage`, `age_at_first_pregnancy`, `district`, `village`, `occupation`, `diet`, `medical_condition`).
- **followups** (`followup_id` PK, `patient_id` FK, `visit_number`, `visit_date`, `blood_pressure_sys`, `blood_pressure_dia`, `weight_kg`).
- **deliveries** (`delivery_id` PK, `patient_id` FK, `date_of_delivery`, `age_at_delivery`, `weight_at_delivery`, `haemoglobin_at_delivery`, `placental_weight`, `term_of_delivery`, `type_of_delivery`).

This separation (3rd normal form) allows storing multiple follow-up visits or deliveries per patient without redundancy. For example, the SQL to create these tables might look like:

```
-- Table: patients
CREATE TABLE patients (
    patient_id VARCHAR(10) PRIMARY KEY,
    height_cm DECIMAL(5,2),
    weight_kg DECIMAL(5,2),
    education_years INT,
    income DECIMAL(10,2),
    age_at_menarche INT,
    age_at_marriage INT,
    age_at_first_pregnancy INT,
    district VARCHAR(100),
    village VARCHAR(100),
    occupation VARCHAR(100),
    diet VARCHAR(100),
    medical_condition VARCHAR(100)
);

-- Table: followups
CREATE TABLE followups (
    followup_id INT AUTO_INCREMENT PRIMARY KEY,
    patient_id VARCHAR(10),
```

```
    visit_number INT,
    visit_date DATE,
    blood_pressure_sys INT,
    blood_pressure_dia INT,
    weight_kg DECIMAL(5,2),
    FOREIGN KEY (patient_id) REFERENCES patients(patient_id)
);

-- Table: deliveries
CREATE TABLE deliveries (
    delivery_id INT AUTO_INCREMENT PRIMARY KEY,
    patient_id VARCHAR(10),
    date_of_delivery DATE,
    age_at_delivery INT,
    weight_at_delivery DECIMAL(5,2),
    haemoglobin_at_delivery DECIMAL(4,2),
    placental_weight DECIMAL(5,2),
    term_of_delivery VARCHAR(50),
    type_of_delivery VARCHAR(50),
    FOREIGN KEY (patient_id) REFERENCES patients(patient_id)
);
```

These tables can be visualized in an ER diagram (entities `Patients`, `FollowUps`, `Deliveries` with one-to-many links). Indexes on `followups.patient_id` and `deliveries.patient_id` (via the foreign key) speed up lookups. The schema SQL is provided in a `schema.sql` deliverable file for easy deployment.

Flask Backend and API Endpoints

We use **Flask** to build a RESTful API server in Python. The app connects to MySQL using a connector library (e.g. `mysql-connector-python` or `PyMySQL`). On startup, we **load initial data** from the Excel files into the database. In code, we can use Pandas' `read_excel` to parse each file into a DataFrame [1], then iterate the rows to insert into the tables. For example:

```
import pandas as pd
import mysql.connector

# Connect to MySQL
db = mysql.connector.connect(user='root', password='password', host='localhost', database='hospit
cursor = db.cursor()

def load_initial_data():
    # Load patients
    df_pat = pd.read_excel('basic_information.xlsx')
    for _, row in df_pat.iterrows():
        cursor.execute("""INSERT INTO patients
            (patient_id, height_cm, weight_kg, education_years, income, age_at_menarche, age_at_m
```

```
            age_at_first_pregnancy, district, village, occupation, diet, medical_condition)
            VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)""",
           (row['patient_id'], row['ht'], row['wt'], row['education'], row['income'],
            row['ageatmenarche'], row['ageatmarriage'], row['ageatfirstpregnancy'],
            row['district'], row['village'], row['occupation'], row['diet'], row['condition']))
    db.commit()

    # Load follow-up visits (flatten wide format)
    df_fup = pd.read_excel('followup_data.xlsx')
    for _, row in df_fup.iterrows():
        pid = row['patient_id']
        # Handle up to 4 visits per patient
        for v in range(1, 5):
            if pd.notna(row[f'Visit{v}_date']):
                cursor.execute("""INSERT INTO followups
                    (patient_id, visit_number, visit_date, blood_pressure_sys, blood_pressure_dia
                    VALUES (%s, %s, %s, %s, %s, %s)""",
                    (pid, v,
                     row[f'Visit{v}_date'], row[f'Visit{v}_bpsys'], row[f'Visit{v}_bpdis'], row[f
    db.commit()

    # Load deliveries
    df_del = pd.read_excel('delivery_information.xlsx')
    for _, row in df_del.iterrows():
        cursor.execute("""INSERT INTO deliveries
            (patient_id, date_of_delivery, age_at_delivery, weight_at_delivery, haemoglobin_at_de
            VALUES (%s, %s, %s, %s, %s, %s, %s, %s)""",
            (row['patient_id'], row['dateofdelivery'], row['ageatdelivery'],
             row['weightatdelivery'], row['haemoglobinatdelivery'], row['placentalweight'],
             row['termofdelivery'], row['typeofdelivery']))
    db.commit()

# Initialize DB schema and data at app startup (if needed)
load_initial_data()
```

Next, we define Flask routes (endpoints) for the required API:

- **GET /api/patient/<id>**: Look up patient by ID. This returns JSON with the patient's basic info, and optionally list of follow-ups and delivery records. For example:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/patient/<patient_id>', methods=['GET'])
def get_patient(patient_id):
    cursor.execute("SELECT * FROM patients WHERE patient_id=%s", (patient_id,))
```

```
    patient = cursor.fetchone()
    if not patient:
        return jsonify({'error': 'Not found'}), 404
    # Convert to dict or list for JSON
    columns = [col[0] for col in cursor.description]
    patient_data = dict(zip(columns, patient))
    return jsonify(patient_data)   # jsonify sets JSON content type automatically
```

- **POST /api/patient**: Insert a new patient record. This expects JSON or form data (from the frontend form). We parse `request.json` (or `request.form`) and insert into `patients`. For example:

```
@app.route('/api/patient', methods=['POST'])
def add_patient():
    data = request.json  # or request.get_json()
    # Example fields: patient_id, height_cm, weight_kg, etc.
    cursor.execute("""INSERT INTO patients
        (patient_id, height_cm, weight_kg, education_years, income, age_at_menarche,
         age_at_marriage, age_at_first_pregnancy, district, village, occupation, diet, medical_co
         VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)""",
        (data['patient_id'], data['height_cm'], data['weight_kg'], data['education_years'], data[
         data['age_at_menarche'], data['age_at_marriage'], data['age_at_first_pregnancy'],
         data['district'], data['village'], data['occupation'], data['diet'], data['medical_condi
    db.commit()
    return jsonify({'status': 'Patient added'}), 201
```

- **Analytics endpoints**: We can create endpoints to run summary queries. For example,
- **GET /api/analytics/delivery_rates** executes the "deliveries per year" query.
- **GET /api/analytics/followup_counts** returns visits per patient.
- **GET /api/analytics/weight_trends** returns average weights over time.

Each route runs a SQL query and returns JSON. For instance:

```
@app.route('/api/analytics/delivery_rates', methods=['GET'])
def delivery_rates():
    cursor.execute("""
        SELECT YEAR(date_of_delivery) AS year, COUNT(*) AS deliveries_count
        FROM deliveries
        GROUP BY YEAR(date_of_delivery)
        ORDER BY year;
    """)
    results = cursor.fetchall()
    return jsonify([{'year': year, 'deliveries': count} for (year,count) in results])
```

Each endpoint uses Flask's `jsonify()` to return JSON (which automatically sets `Content-Type: application/json` [2]). These routes form a simple REST API (GET for lookup/analytics, POST for inserting new data).

Integrating the React Frontend

The existing React frontend can call these endpoints. For example, the patient data entry form in React should submit to `/api/patient` using `fetch` or `axios`:

- **In React**: When the user submits the form, gather the form state and send a POST request, e.g.:

```
fetch('/api/patient', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(newPatientData)
})
.then(res => res.json())
.then(response => console.log(response));
```

  Ensure CORS or proxy is configured if frontend is on a different origin. The Flask app should handle this POST and insert the new patient into MySQL. After insertion, the React app can update the UI (e.g. navigate to a patient profile or show success message).

- **Fetching Data**: Similarly, to display patient info, the React app can call `GET /api/patient/<id>` and render the JSON data in the UI. Analytics summaries can be fetched from `/api/analytics/...` and displayed (e.g., as charts or tables).

This way the backend provides a clear JSON API, and the React frontend simply consumes it. The integration logic primarily involves calling the correct endpoint on form submission and handling responses.

Analytics SQL Queries

Below are example optimized SQL queries for the requested analytics:

1. **Delivery counts per year:** Calculate number of deliveries by year.

```
SELECT YEAR(date_of_delivery) AS year,
       COUNT(*) AS deliveries_count
FROM deliveries
GROUP BY YEAR(date_of_delivery)
ORDER BY year;
```

   Insight: Shows how many deliveries occurred each year.

2. **Follow-up visit frequency per patient:** Count of visits each patient has.

```
SELECT patient_id,
       COUNT(*) AS visit_count
FROM followups
GROUP BY patient_id
ORDER BY visit_count DESC;
```

Insight: Identifies patients with most follow-ups (e.g. for high-risk monitoring).

3. **Average patient weight over time:** Trend of average weight (from follow-ups) by year.

```
SELECT YEAR(visit_date) AS year,
       ROUND(AVG(weight_kg),2) AS avg_weight
FROM followups
GROUP BY YEAR(visit_date)
ORDER BY year;
```

Insight: Shows how patient weights (on average) change year over year.

(These queries are provided in a `analytics_queries.sql` *script.)*

README / Summary

- **Database Design:** We created a normalized schema with `patients`, `followups`, and `deliveries` tables. Each patient can have many follow-up records and deliveries, linked by the patient ID (PK/FK). Fields from the Excel files map directly into these tables. All tables use appropriate keys and data types (e.g. `DATE` for dates, `DECIMAL` for weights).

- **API Structure:** The Flask app exposes RESTful endpoints. Key routes are:

- `GET /api/patient/<id>` – retrieves a patient record (returns JSON).
- `POST /api/patient` – adds a new patient (accepts JSON form data).
- `GET /api/analytics/...` – runs summary queries for analytics. Flask's `jsonify()` is used to send JSON responses easily [2].

- **Data Loading:** On startup, the Flask app uses Pandas to read the `.xlsx` files and populate MySQL tables. The code uses `pd.read_excel(...)` to parse each file into a DataFrame [1], then loops through rows to insert into the database. Follow-up data (originally in wide format) is normalized into separate rows per visit.

- **Frontend Integration:** The existing React frontend can call these Flask endpoints. For example, the patient entry form should submit via a POST fetch request to `/api/patient` (JSON body).

Similarly, components can fetch from `/api/patient/:id` to display patient info, and from `/api/analytics/...` to show reports (e.g. plotting delivery counts per year). This clean separation lets the React UI handle user interaction while Flask/MySQL manage data storage and logic.

- **Analytics Insights:** Our SQL queries produce quick summaries:

- *Delivery rates per year* – useful for seeing trends in number of births each year.
- *Follow-up counts per patient* – highlights patient visit frequency.
- *Average weight over time* – shows weight trends, which could correlate with health interventions.

These insights can feed into the dashboard's charts or tables, providing actionable information for hospital planning.

References: We use standard libraries like Pandas (`read_excel`) [1] and Flask (`jsonify`) [2] to implement these features efficiently.

---

1 **python - How to read an excel file in flask webserver memory at the start of flask app? - Stack Overflow**
https://stackoverflow.com/questions/70483997/how-to-read-an-excel-file-in-flask-webserver-memory-at-the-start-of-flask-app

2 **Use jsonify() instead of json.dumps() in Flask | GeeksforGeeks**
https://www.geeksforgeeks.org/use-jsonify-instead-of-json-dumps-in-flask/