```
{
"id": "1",
"email": "knandini7816@gmail.com",
"full_name": "Nandini",
"age": 19,
"profession": "student",
"patient_data": {
"current_diagnosis": [
"Thyroid"
],
"medications": [
"Thyroxine 100mg"
],
"dietary_preferences": [
"Non Veg"
],
"exercise_routine": [
"Daily"
],
"health_goals": [
"Loose Weight"
],
"current_symptoms": [
"Drousy"
]
}
}
```


`users` Collection

Stores user details and acts as the central reference for other collections.


```
{
"_id": "ObjectId",
"email": "string",
"fullName": "string",
"phone": "string",
"age": "number",
"profession": "string"
}
```


`patient_data` Collection

Contains detailed medical information for each user.

{
"_id": "ObjectId",
"userId": "ObjectId", // Reference to `users._id`
"currentDiagnosis": ["string"],
"medications": ["string"],
"dietaryPreferences": ["string"],
"exerciseRoutine": ["string"],
"healthGoals": ["string"],
"currentSymptoms": ["string"]
}

`food_logs` Collection

Stores daily food logs for each user.

{
"_id": "ObjectId",
"userId": "ObjectId", // Reference to `users._id`
"date": "date",
"meals": {
"breakfast": ["string"],
"snack": ["string"],
"lunch": ["string"],
"eveningSnack": ["string"],
"dinner": ["string"]
},
"totalCalories": "number"
}

`exercise_logs` Collection

Tracks exercise activities for each user.

{
"_id": "ObjectId",
"userId": "ObjectId", // Reference to `users._id`
"date": "date",
"exercises": [
{
"exerciseType": "string",
"durationMinutes": "number",
"caloriesBurned": "number",
"targetCalories": "number"

}
]
}

`medicines` Collection

Stores medication information for each user.

{
"_id": "ObjectId",
"userId": "ObjectId", // Reference to `users._id`
"prescriptions": [
{
"medicineName": "string",
"quantity": "number",
"dosage": "string",
"frequency": "string",
"startDate": "date",
"endDate": "date"
}
]
}

`water_logs` Collection

Tracks water intake for each user.

{
"_id": "ObjectId",
"userId": "ObjectId", // Reference to `users._id`
"date": "date",
"waterIntake": {
"quantity": "number", // e.g., in ml
"unit": "string" // e.g., "ml", "liters", "glasses"
}
}

`reports` Collection

Stores healthcare report data.

{
"_id": "ObjectId",
"userId": "ObjectId", // Reference to `users._id`
"reportType": "string",
"fileUrl": "string",

"uploadedAt": "date",
"analysisResults": {
"keyMetric1": "number",
"keyMetric2": "number"
}
}

`community` Collection

Handles community posts and queries.

Here’s the revised backend structure and code. This version embeds `patient_data` directly within the `users` collection, simplifying data retrieval and ensuring all patient-specific data remains associated with the user. Other logs (food, exercise, etc.) remain separate but reference the `user_id` for scalability.

Updated Backend Structure

```
backend/
├── models/
│   ├── __init__.py
│   ├── user_model.py
│   ├── food_logs_model.py
│   ├── exercise_logs_model.py
│   ├── medicines_model.py
│   ├── water_logs_model.py
│   ├── reports_model.py
├── routes/
│   ├── __init__.py
│   ├── user_routes.py
│   ├── food_logs_routes.py
│   ├── exercise_logs_routes.py
│   ├── medicines_routes.py
│   ├── water_logs_routes.py
│   ├── reports_routes.py
├── main.py
├── database.py
├── requirements.txt
```

---

models/user_model.py

```
from pydantic import BaseModel, EmailStr

from typing import List, Optional

from datetime import datetime


class PatientData(BaseModel):

    current_diagnosis: List[str]

    medications: List[str]

    dietary_preferences: List[str]

    exercise_routine: List[str]

    health_goals: List[str]

    current_symptoms: List[str]


class User(BaseModel):

    id: Optional[str]  # MongoDB will auto-generate this if not provided

    email: EmailStr

    full_name: str

    phone: str

    age: int

    profession: str

    patient_data: Optional[PatientData]

    created_at: Optional[datetime]  # Automatically set current date when
creating a user
```

---

models/food_logs_model.py

```python
from pydantic import BaseModel
from typing import Dict, List, Optional
from datetime import datetime


class FoodLogs(BaseModel):
    id: Optional[str]
    user_id: str
    date: datetime  # Using datetime instead of str
    meals: Dict[str, List[str]]
    total_calories: int
```

---

models/exercise_logs_model.py

```python
from pydantic import BaseModel
from typing import List


class ExerciseLogEntry(BaseModel):
    exercise_type: str
    duration_minutes: int
    calories_burned: int
    target_calories: int


class ExerciseLogs(BaseModel):
    id: str | None
    user_id: str
```

```
    date: str

    exercises: List[ExerciseLogEntry]
```

---

models/medicines_model.py

```
from pydantic import BaseModel

from typing import List


class Prescription(BaseModel):

    medicine_name: str

    quantity: int

    dosage: str

    frequency: str

    start_date: str

    end_date: str


class Medicines(BaseModel):

    id: str | None

    user_id: str

    prescriptions: List[Prescription]
```

---

models/water_logs_model.py

```
from pydantic import BaseModel
```

```python
class WaterLogs(BaseModel):

    id: str | None

    user_id: str

    date: str

    water_intake: dict  # {"quantity": int, "unit": str}
```

---

models/reports_model.py

```python
from pydantic import BaseModel

from typing import Optional

from datetime import date, datetime



class Reports(BaseModel):

    user_id: str

    report_title: str

    report_content: str

    date_created: date  # This is the input type for date



    # Convert date to datetime before storing in MongoDB

    def to_mongo(self):

        data = self.dict()

        data["date_created"] = datetime.combine(self.date_created,
datetime.min.time())

        return data
```

```
    class Config:

        orm_mode = True  # To allow MongoDB's data to work with Pydantic
models
```

---

database.py

```
from motor.motor_asyncio import AsyncIOMotorClient
from dotenv import load_dotenv
import os

# Load environment variables from .env file
load_dotenv()

# Retrieve the MONGO_URI from the environment variables
MONGO_URI = os.getenv("MONGO_URI")

# Initialize the MongoDB client
client = AsyncIOMotorClient(MONGO_URI)
db = client["nurture_sync"]
```

---

main.py

```
from fastapi import FastAPI

from routes.user_routes import router as user_router

from routes.food_logs_routes import router as food_logs_router

from routes.exercise_logs_routes import router as exercise_logs_router

from routes.medicines_routes import router as medicines_router

from routes.water_logs_routes import router as water_logs_router

from routes.reports_routes import router as reports_router



app = FastAPI()
```

```
# Including routers with correct prefix

app.include_router(user_router, prefix="/api/users", tags=["Users"])

app.include_router(food_logs_router, prefix="/api/food_logs", tags=["Food
Logs"])

app.include_router(exercise_logs_router, prefix="/api/exercise_logs", tags=
["Exercise Logs"])

app.include_router(medicines_router, prefix="/api/medicines", tags=
["Medicines"])

app.include_router(water_logs_router, prefix="/api/water_logs", tags=["Water
Logs"])

app.include_router(reports_router, prefix="/api/reports", tags=["Reports"])



@app.get("/")

async def read_root():

    return {"message": "Welcome to the NS API!"}
```

`requirements.txt`

```
fastapi
motor
pydantic
uvicorn
```

`routes/food_logs_routes.py`

```
from fastapi import APIRouter, HTTPException

from bson import ObjectId  # Import ObjectId to handle MongoDB's _id

from models.food_logs_model import FoodLogs
```

```python
from database import db


router = APIRouter()


# Helper function to validate ObjectId
def is_valid_object_id(id_str: str) -> bool:
    try:
        ObjectId(id_str)
        return True
    except Exception:
        return False


# Create Food Log
@router.post("/")
async def create_food_log(log: FoodLogs):
    log_dict = log.dict()
    result = await db["food_logs"].insert_one(log_dict)
    return {"id": str(result.inserted_id)}


# Get Food Logs by user_id
@router.get("/{user_id}")
async def get_food_logs(user_id: str):
    # Validate if the user_id is a valid ObjectId
    if not is_valid_object_id(user_id):
        raise HTTPException(status_code=400, detail="Invalid user_id
format")
```

```
    try:
        # Convert user_id to ObjectId
        user_id = ObjectId(user_id)
        # Query the database
        logs = await db["food_logs"].find({"user_id": str(user_id)}).to_list(length=100)
        # Check if logs exist
        if not logs:
            raise HTTPException(status_code=404, detail="Food logs not found")
        # Format the logs for output
        for log in logs:
            log["_id"] = str(log["_id"])  # Convert ObjectId to string
        return logs
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error fetching food logs: {str(e)}")


# Update Food Log by log_id
@router.put("/{log_id}")
async def update_food_log(log_id: str, updated_log: FoodLogs):
    try:
        log_id = ObjectId(log_id)  # Convert log_id to ObjectId
    except Exception:
        raise HTTPException(status_code=400, detail="Invalid log ID format")
```

```
    result = await db["food_logs"].update_one(
        {"_id": log_id},
        {"$set": updated_log.dict()}
    )

    if result.matched_count == 0:
        raise HTTPException(status_code=404, detail="Log not found")

    return {"detail": "Food log updated successfully"}

# Delete Food Log by log_id
@router.delete("/{log_id}")
async def delete_food_log(log_id: str):
    try:
        log_id = ObjectId(log_id)  # Convert log_id to ObjectId
    except Exception:
        raise HTTPException(status_code=400, detail="Invalid log ID format")

    result = await db["food_logs"].delete_one({"_id": ObjectId(log_id)})

    if result.deleted_count == 0:
        raise HTTPException(status_code=404, detail="Log not found")

    return {"detail": "Food log deleted successfully"}
```

routes/user_routes.py

```python
from fastapi import APIRouter, HTTPException

from database import db

from models.user_model import User

from bson import ObjectId


router = APIRouter()


@router.post("/")

async def create_user(user: User):

    user_dict = user.dict()

    result = await db["users"].insert_one(user_dict)

    return {"id": str(result.inserted_id)}


@router.get("/{user_id}")

async def get_user(user_id: str):

    user = await db["users"].find_one({"_id": ObjectId(user_id)})

    if not user:

        raise HTTPException(status_code=404, detail="User not found")

    user["_id"] = str(user["_id"])

    return user


@router.put("/{user_id}")

async def update_user(user_id: str, updated_user: User):
```

```
    result = await db["users"].update_one(
        {"_id": ObjectId(user_id)},
        {"$set": updated_user.dict()}
    )
    if result.matched_count == 0:
        raise HTTPException(status_code=404, detail="User not found")
    return {"detail": "User updated successfully"}


@router.delete("/{user_id}")
async def delete_user(user_id: str):
    result = await db["users"].delete_one({"_id": ObjectId(user_id)})
    if result.deleted_count == 0:
        raise HTTPException(status_code=404, detail="User not found")
    return {"detail": "User deleted successfully"}
```

`routes/exercise_logs_routes.py`

```python
from fastapi import APIRouter, HTTPException
from bson import ObjectId  # Import ObjectId to handle MongoDB's _id
from models.exercise_logs_model import ExerciseLogs
from database import db

router = APIRouter()

# Helper function to validate ObjectId
```

```python
def is_valid_object_id(id_str: str) -> bool:
    try:
        ObjectId(id_str)
        return True
    except Exception:
        return False


# Create Exercise Log
@router.post("/")
async def create_exercise_log(log: ExerciseLogs):
    log_dict = log.dict()
    result = await db["exercise_logs"].insert_one(log_dict)
    return {"id": str(result.inserted_id)}


# Get Exercise Logs by user_id
@router.get("/{user_id}")
async def get_exercise_logs(user_id: str):
    if not is_valid_object_id(user_id):
        raise HTTPException(status_code=400, detail="Invalid user_id
format")


    try:
        user_id = ObjectId(user_id)
        logs = await db["exercise_logs"].find({"user_id":
str(user_id)}).to_list(length=100)
        if not logs:
```

```
            raise HTTPException(status_code=404, detail="Exercise logs not
found")



        for log in logs:

            log["_id"] = str(log["_id"])

        return logs

    except Exception as e:

        raise HTTPException(status_code=500, detail=f"Error fetching
exercise logs: {str(e)}")



# Update Exercise Log by log_id

@router.put("/{log_id}")

async def update_exercise_log(log_id: str, updated_log: ExerciseLogs):

    try:

        log_id = ObjectId(log_id)

    except Exception:

        raise HTTPException(status_code=400, detail="Invalid log ID format")



    result = await db["exercise_logs"].update_one(

        {"_id": log_id},

        {"$set": updated_log.dict()}

    )



    if result.matched_count == 0:

        raise HTTPException(status_code=404, detail="Log not found")
```

```
    return {"detail": "Exercise log updated successfully"}


# Delete Exercise Log by log_id
@router.delete("/{log_id}")
async def delete_exercise_log(log_id: str):
    try:
        log_id = ObjectId(log_id)
    except Exception:
        raise HTTPException(status_code=400, detail="Invalid log ID format")

    result = await db["exercise_logs"].delete_one({"_id": ObjectId(log_id)})

    if result.deleted_count == 0:
        raise HTTPException(status_code=404, detail="Log not found")

    return {"detail": "Exercise log deleted successfully"}
```

---

routes/medicines_routes.py

```
from fastapi import APIRouter, HTTPException
from bson import ObjectId  # Import ObjectId to handle MongoDB's _id
from models.medicines_model import Medicines
from database import db

router = APIRouter()
```

```python
# Helper function to validate ObjectId
def is_valid_object_id(id_str: str) -> bool:
    try:
        ObjectId(id_str)
        return True
    except Exception:
        return False

# Create Medicines Record
@router.post("/")
async def create_medicines(medicines: Medicines):
    medicines_dict = medicines.dict()
    result = await db["medicines"].insert_one(medicines_dict)
    return {"id": str(result.inserted_id)}

# Get Medicines by user_id
@router.get("/{user_id}")
async def get_medicines(user_id: str):
    if not is_valid_object_id(user_id):
        raise HTTPException(status_code=400, detail="Invalid user_id
format")

    try:
        user_id = ObjectId(user_id)
        medicines = await db["medicines"].find({"user_id":
```

```
str(user_id)}).to_list(length=100)

        if not medicines:

            raise HTTPException(status_code=404, detail="Medicines not
found")



        for medicine in medicines:

            medicine["_id"] = str(medicine["_id"])

        return medicines

    except Exception as e:

        raise HTTPException(status_code=500, detail=f"Error fetching
medicines: {str(e)}")



# Update Medicines Record by record_id

@router.put("/{record_id}")

async def update_medicines(record_id: str, updated_medicines: Medicines):

    try:

        record_id = ObjectId(record_id)

    except Exception:

        raise HTTPException(status_code=400, detail="Invalid record ID
format")



    result = await db["medicines"].update_one(

        {"_id": record_id},

        {"$set": updated_medicines.dict()}

    )



    if result.matched_count == 0:
```

```
        raise HTTPException(status_code=404, detail="Record not found")

    return {"detail": "Medicines record updated successfully"}

# Delete Medicines Record by record_id
@router.delete("/{record_id}")
async def delete_medicines(record_id: str):
    try:
        record_id = ObjectId(record_id)
    except Exception:
        raise HTTPException(status_code=400, detail="Invalid record ID
format")

    result = await db["medicines"].delete_one({"_id": ObjectId(record_id)})

    if result.deleted_count == 0:
        raise HTTPException(status_code=404, detail="Record not found")

    return {"detail": "Medicines record deleted successfully"}
```

`routes/water_logs_routes.py`

```python
from fastapi import APIRouter, HTTPException
from bson import ObjectId  # Import ObjectId to handle MongoDB's _id
from models.water_logs_model import WaterLogs
from database import db
```

```python
router = APIRouter()


# Helper function to validate ObjectId
def is_valid_object_id(id_str: str) -> bool:
    try:
        ObjectId(id_str)
        return True
    except Exception:
        return False


# Create Water Log
@router.post("/")
async def create_water_log(log: WaterLogs):
    log_dict = log.dict()
    result = await db["water_logs"].insert_one(log_dict)
    return {"id": str(result.inserted_id)}


# Get Water Logs by user_id
@router.get("/{user_id}")
async def get_water_logs(user_id: str):
    if not is_valid_object_id(user_id):
        raise HTTPException(status_code=400, detail="Invalid user_id
format")
```

```
    try:
        user_id = ObjectId(user_id)
        logs = await db["water_logs"].find({"user_id": str(user_id)}).to_list(length=100)
        if not logs:
            raise HTTPException(status_code=404, detail="Water logs not found")

        for log in logs:
            log["_id"] = str(log["_id"])
        return logs
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error fetching water logs: {str(e)}")

# Update Water Log by log_id
@router.put("/{log_id}")
async def update_water_log(log_id: str, updated_log: WaterLogs):
    try:
        log_id = ObjectId(log_id)
    except Exception:
        raise HTTPException(status_code=400, detail="Invalid log ID format")

    result = await db["water_logs"].update_one(
        {"_id": log_id},
        {"$set": updated_log.dict()}
    )
```

```
    if result.matched_count == 0:
        raise HTTPException(status_code=404, detail="Log not found")

    return {"detail": "Water log updated successfully"}

# Delete Water Log by log_id
@router.delete("/{log_id}")
async def delete_water_log(log_id: str):
    try:
        log_id = ObjectId(log_id)
    except Exception:
        raise HTTPException(status_code=400, detail="Invalid log ID format")

    result = await db["water_logs"].delete_one({"_id": ObjectId(log_id)})

    if result.deleted_count == 0:
        raise HTTPException(status_code=404, detail="Log not found")

    return {"detail": "Water log deleted successfully"}
```

---

routes/reports_routes.py

```
from fastapi import APIRouter, HTTPException
from bson import ObjectId  # Import ObjectId to handle MongoDB's _id
```

```python
from models.reports_model import Reports  # Import the Reports model
from database import db


router = APIRouter()


# Helper function to validate ObjectId
def is_valid_object_id(id_str: str) -> bool:
    try:
        ObjectId(id_str)
        return True
    except Exception:
        return False


# Create Report
@router.post("/")
async def create_report(report: Reports):
    try:
        report_dict = report.to_mongo()  # Convert the report using the to_mongo method
        result = await db["reports"].insert_one(report_dict)  # Insert the report into the MongoDB collection
        return {"id": str(result.inserted_id)}  # Return the inserted report ID
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error creating report: {str(e)}")  # Log the error
```

```python
# Get Reports by user_id

@router.get("/{user_id}")

async def get_reports(user_id: str):

    if not is_valid_object_id(user_id):

        raise HTTPException(status_code=400, detail="Invalid user_id
format")



    try:

        user_id = ObjectId(user_id)

        reports = await db["reports"].find({"user_id":
str(user_id)}).to_list(length=100)

        if not reports:

            raise HTTPException(status_code=404, detail="Reports not found")



        for report in reports:

            report["_id"] = str(report["_id"])  # Convert ObjectId to string

        return reports

    except Exception as e:

        raise HTTPException(status_code=500, detail=f"Error fetching
reports: {str(e)}")



@router.put("/{report_id}")

async def update_report(report_id: str, updated_report: Reports):

    try:

        report_id = ObjectId(report_id)  # Convert report_id to ObjectId

    except Exception:
```

```
        raise HTTPException(status_code=400, detail="Invalid report ID
format")



    # Convert updated report to the correct format

    updated_report_dict = updated_report.to_mongo()  # Convert the
date_created field properly

    result = await db["reports"].update_one(

        {"_id": report_id},

        {"$set": updated_report_dict}  # Use the correctly formatted report
data

    )



    if result.matched_count == 0:

        raise HTTPException(status_code=404, detail="Report not found")



    return {"detail": "Report updated successfully"}



# Delete Report by report_id

@router.delete("/{report_id}")

async def delete_report(report_id: str):

    try:

        report_id = ObjectId(report_id)  # Convert report_id to ObjectId

    except Exception:

        raise HTTPException(status_code=400, detail="Invalid report ID
format")



    result = await db["reports"].delete_one({"_id": ObjectId(report_id)})
```

```
    if result.deleted_count == 0:
        raise HTTPException(status_code=404, detail="Report not found")

    return {"detail": "Report deleted successfully"}
```

---

here is the flow signup and login are handled by firebase and rest of the data of health matrices of the user will be stored in mongodb. So while signing up the user deatils will be stored in mongodb as well, then a set of questionarie will be asked(with multiple pages after signup), that is stored in patient_data the questions are the cols of that model,after that info of the user is retrived and profile is created, user fills the questionarie and then the data of it is stored in another collection patient_data collection in nurture_sync db , how to store the data , how to embed or link it ? how to map user's food, medication, exercise all details to the user all the collections except community are in nurture_sync db, nurture_sync Collections:
exercise_logs , food_logs ,medicines, patient_data , reports,users,water_logs

The routes in `main.py` define the URLs that the backend API provides. Each route corresponds to specific functionality or operations in the application. These routes are registered using **FastAPI's router system**, which organizes related functionality into modules for better modularity and scalability.

How Routes Work

1. **Router Definition:**
   - Each route file (e.g., `user_routes.py`, `food_logs_routes.py`) defines endpoints related to specific functionality.
   - A router groups these endpoints together. For example, `user_routes.router` contains all the routes for user-related operations.
2. **Route Registration in `main.py`:**
   - The `app.include_router()` function in `main.py` registers these routes with the main FastAPI application (`app`).
   - Each router is given a **prefix** (e.g., `/api/users`) and optional **tags** (used for API documentation).
3. **How the URL Structure Works:**

- A route's full URL is constructed by combining the base URL of the API with the prefix and the route path.
- Example:
  - Base URL: `http://localhost:8000`
  - Prefix: `/api/users`
  - Route Path: `/users/{user_id}`
  - Full URL: `http://localhost:8000/api/users/{user_id}`

Example Routes and Their Usage

Here's what happens for each router included in `main.py`:

1. Users Routes

Defined in: `user_routes.py`
Prefix: `/api/users`
Example Endpoints:

- `POST /api/users/`: Creates a new user in MongoDB.
- `GET /api/users/{user_id}`: Fetches details of a specific user by their ID.
- `PUT /api/users/{user_id}`: Updates user information.
- `DELETE /api/users/{user_id}`: Deletes a user.

Usage in Browser:

- Visiting `http://localhost:8000/api/users/` directly in the browser will not display anything meaningful because this is a POST endpoint and requires a request body (JSON payload).
- You could use tools like **Postman** or **cURL** to interact with these endpoints.

---

2. Food Logs Routes

Defined in: `food_logs_routes.py`
Prefix: `/api/food_logs`
Example Endpoints:

- `POST /api/food_logs/`: Adds a food log for a user.
- `GET /api/food_logs/{user_id}`: Fetches all food logs for a specific user.
- `PUT /api/food_logs/{log_id}`: Updates a food log.
- `DELETE /api/food_logs/{log_id}`: Deletes a food log.

3. Exercise Logs Routes

Defined in: `exercise_logs_routes.py`
Prefix: `/api/exercise_logs`
Example Endpoints:

- `POST /api/exercise_logs/` : Adds an exercise log for a user.
- `GET /api/exercise_logs/{user_id}` : Fetches all exercise logs for a specific user.
- `PUT /api/exercise_logs/{log_id}` : Updates an exercise log.
- `DELETE /api/exercise_logs/{log_id}` : Deletes an exercise log.

---

4. Medicines Routes

Defined in: `medicines_routes.py`
Prefix: `/api/medicines`
Example Endpoints:

- `POST /api/medicines/` : Adds medicine information for a user.
- `GET /api/medicines/{user_id}` : Fetches all medicines for a specific user.
- `PUT /api/medicines/{medicine_id}` : Updates a medicine entry.
- `DELETE /api/medicines/{medicine_id}` : Deletes a medicine entry.

---

5. Water Logs Routes

Defined in: `water_logs_routes.py`
Prefix: `/api/water_logs`
Example Endpoints:

- `POST /api/water_logs/` : Adds water intake information for a user.
- `GET /api/water_logs/{user_id}` : Fetches all water intake logs for a specific user.
- `PUT /api/water_logs/{log_id}` : Updates a water log.
- `DELETE /api/water_logs/{log_id}` : Deletes a water log.

---

6. Reports Routes

Defined in: `reports_routes.py`
Prefix: `/api/reports`

Example Endpoints:

- `POST /api/reports/` : Uploads a new report for a user.
- `GET /api/reports/{user_id}` : Fetches all reports for a specific user.
- `PUT /api/reports/{report_id}` : Updates a report.
- `DELETE /api/reports/{report_id}` : Deletes a report.

What is Displayed in the Browser?

1. **Base URL (`http://localhost:8000/`):**
   - The `root()` function in `main.py` handles the root URL `/`.
   - When you visit `http://localhost:8000` in a browser, you'll see:

     ```
     {
       "message": "Welcome to the NurtureSync API!"
     }
     ```

2. **Interactive API Documentation:** FastAPI automatically generates interactive API documentation at:

   - **Swagger UI:** `http://localhost:8000/docs` (default endpoint for API documentation)
   - **ReDoc:** `http://localhost:8000/redoc` (alternative documentation)

   What You'll See:
   - A list of all available endpoints grouped by tags (e.g., Users, Food Logs).
   - Detailed information about each endpoint, including request parameters, request body, and responses.
   - An interactive interface where you can test the endpoints directly.
3. **Specific Endpoints (`/api/...`):**
   - For `GET` endpoints (e.g., `/api/users/{user_id}`), you can directly visit them in the browser by replacing `{user_id}` with an actual value. If the route is correctly set up and data exists, you'll see the JSON response.

How to Test These Routes?

- Use **Swagger UI** at `/docs` for testing without additional tools.
- Use **Postman** or **cURL** for advanced testing or automation.

- Use **JavaScript (e.g., Fetch API)** or Flutter's HTTP package to integrate with the frontend.

Let me know if you'd like a walkthrough of any specific functionality!

1. Use a Unique Identifier for Mapping

- The `user_id` field in all models should serve as the foreign key that links the `users` collection with other collections like `patient_data`, `food_logs`, `exercise_logs`, etc.
- This ensures that all records related to a user can be easily queried using their `user_id`.

Frontend Integration

- Pass `user_id` as part of API requests from Flutter to uniquely identify the user.
- Save the `user_id` securely in local storage after login/signup for reuse.

Summary

- Use `user_id` to link all data across collections.
- Embed tightly coupled data, reference loosely coupled or large datasets.
- Adjust backend models and routes to consistently handle `user_id`.
- Ensure API requests from Flutter always include the `user_id`.