FitFlow — Schema Modeling & Frontend-Backend Integration

Date: 2025-11-01 Goal: Provide concrete DB schemas (Mongoose), detailed API contracts, TypeScript interfaces for the frontend, example queries/aggregations, and integration instructions (auth, fetch client, SWR/React Query). Use this as the single-source contract when building the Express + Mongo backend.

---

1. Summary / Contracts

This document is the contract between frontend and backend. It contains:

- Mongoose schema designs (models and indexes)
- API endpoints and request/response JSON shapes
- Frontend TypeScript interfaces for responses and payloads
- Auth cookie vs Bearer guidance and example client code
- Common aggregations for analytics
- Edge cases, validation and migration notes

Design goals:

- Clear, minimal models to start; support denormalization where performance demands it
- Read-friendly JSON shapes for the frontend
- Predictable error envelope

Key Requirements:

- **Diet Plans:** Generated DAILY based on previous day's diet progress + user requirements. Previous day data archived to monthly diet report.
- **Workout Plans:** Generated for a duration (e.g., 4-week cycle) based on user input. When regenerating next cycle, use previous month's workout progress report.
- **Monthly Reports:** Aggregate daily diet logs and workout progress into monthly summaries for analysis and future plan generation.

---

2. Error & Response Envelope

All successful responses:

```
{ "ok": true, "data": <any> }
```

All errors:

```
{ "ok": false, "error": { "code": "SOME_CODE", "message": "Human friendly
message", "details?: any } }
```

Status codes: 200 (OK), 201 (Created), 204 (No Content), 400 (Bad Request), 401 (Unauthorized), 403 (Forbidden), 404 (Not Found), 429 (Too Many Requests), 500 (Server Error).

---

3. Mongoose Models (detailed)

Notes: use TypeScript types + mongoose schemas. Use timestamps: true for createdAt/updatedAt where useful.

3.1. User

Filename: `models/User.ts`

Key fields and rationale:

- Keep `email` unique and indexed
- Store `passwordHash` only for local auth
- `role` for admin/user RBAC
- `profile` object for measurements and goals
- sessions/refresh tokens stored in separate collection (Session)

Mongoose schema (TypeScript-like):

```
import { Schema, model } from 'mongoose';

const UserSchema = new Schema({
  email: { type: String, required: true, unique: true, index: true },
  passwordHash: { type: String },
  name: { type: String, required: true },
  role: { type: String, enum: ['user','admin'], default: 'user', index:
true },
  profile: {
    age: Number,
    weight: Number,
    height: Number,
    gender: { type: String, enum: ['male','female','other'] },
    goals: [String]
  },
  subscription: {
    plan: String,
    status: { type: String, enum: ['active','inactive'], default:
'inactive' },
    expiresAt: Date
  }
}, { timestamps: true });

export default model('User', UserSchema);
```

Indexes:

- `{ email: 1 }` unique
- `{ role: 1 }`

Frontend interface (types/users.ts):

```
export interface UserProfile { age?: number; weight?: number; height?:
number; gender?: string; goals?: string[] }
export interface UserDTO { id: string; email: string; name: string; role:
'user'|'admin'; profile?: UserProfile; subscription?: { plan?: string;
status?: string; expiresAt?: string } }
```

3.2. WorkoutPlan

Filename: `models/WorkoutPlan.ts`

Rationale: each plan belongs to a user and is generated for a specific duration (e.g., 4 weeks, 8 weeks). Track when the plan is active and its cycle.

```
const ExerciseSchema = new Schema({
  name: { type: String, required: true },
  sets: { type: Number, default: 3 },
  reps: { type: String, default: '8-12' },
  rest: { type: Number, default: 60 }, // seconds
  notes: String
});

const DaySchema = new Schema({ day: { type: String }, exercises:
[ExerciseSchema] });

const WorkoutPlanSchema = new Schema({
  userId: { type: Schema.Types.ObjectId, ref: 'User', index: true,
required: true },
  name: { type: String, default: 'Workout Cycle' },
  duration: { type: Number, required: true }, // duration in weeks (e.g.,
4, 8, 12)
  startDate: { type: Date, required: true },
  endDate: { type: Date, required: true },
  status: { type: String, enum: ['active', 'completed', 'cancelled'],
default: 'active', index: true },
  days: [DaySchema],
  generatedFrom: { type: Schema.Types.ObjectId, ref:
'MonthlyWorkoutReport' } // reference to previous month report if
regenerated
}, { timestamps: true });

export default model('WorkoutPlan', WorkoutPlanSchema);
```

Front DTO:

```
export interface ExerciseDTO { name: string; sets: number; reps: string;
rest: number; notes?: string }
export interface DayDTO { day: string; exercises: ExerciseDTO[] }
export interface WorkoutPlanDTO { id: string; userId: string; name:
string; duration: number; startDate: string; endDate: string; status:
string; days: DayDTO[]; generatedFrom?: string }
```

3.3. DietPlan

Filename: `models/DietPlan.ts`

Rationale: Diet plans are generated DAILY based on previous day's progress. Track the date, previous day link, and generation source.

```
const FoodSchema = new Schema({ name: String, portion: String, calories:
Number, macros: { p: Number, c: Number, f: Number } });
const MealSchema = new Schema({ name: String, time: String, calories:
Number, foods: [FoodSchema] });

const DietPlanSchema = new Schema({
  userId: { type: Schema.Types.ObjectId, ref: 'User', index: true,
required: true },
  name: String,
  date: { type: Date, required: true, index: true }, // specific day this
plan is for
  dailyCalories: Number,
  macros: { protein: Number, carbs: Number, fats: Number },
  meals: [MealSchema],
  generatedFrom: { type: String, enum: ['manual', 'ai', 'auto-daily'],
default: 'manual' },
  previousDayProgressId: { type: Schema.Types.ObjectId, ref: 'ProgressLog'
}, // link to previous day's progress
  notes: String
}, { timestamps: true });

// Compound index for efficient daily plan lookup
DietPlanSchema.index({ userId: 1, date: 1 });

export default model('DietPlan', DietPlanSchema);
```

DTO:

```
export interface FoodDTO { name: string; portion?: string; calories?:
number; macros?: { p?: number; c?: number; f?: number } }
export interface MealDTO { name: string; time?: string; calories?: number;
```

```
foods: FoodDTO[] }
export interface DietPlanDTO { id: string; userId: string; name?: string;
date: string; dailyCalories?: number; macros?: { protein:number;
carbs:number; fats:number }; meals: MealDTO[]; generatedFrom: string;
previousDayProgressId?: string; notes?: string }
```

3.4. ProgressLog

Filename: `models/ProgressLog.ts`

Store daily progress per user; keep granular logs for workouts and meals.

```
const LoggedMealSchema = new Schema({ mealName: String, loggedAt: Date,
calories: Number, macros: { p: Number, c: Number, f: Number } });
const WorkoutProgressSchema = new Schema({ day: String,
completedExercises: Number, totalExercises: Number, durationSec: Number
});

const ProgressLogSchema = new Schema({
  userId: { type: Schema.Types.ObjectId, ref: 'User', index: true,
required: true },
  date: { type: Date, default: () => new Date(), index: true },
  workout: WorkoutProgressSchema,
  meals: [LoggedMealSchema]
}, { timestamps: true });

export default model('ProgressLog', ProgressLogSchema);
```

Compound index recommendation: `{ userId: 1, date: -1 }` to quickly fetch timelines.

DTO:

```
export interface ProgressLogDTO { id: string; userId: string; date:
string; workout?: { day?: string; completedExercises?: number;
totalExercises?: number; durationSec?: number }; meals?: Array<{
mealName?: string; loggedAt?: string; calories?: number; macros?: { p?:
number; c?: number; f?: number } }> }
```

3.5. Session / RefreshToken (optional)

Filename: `models/Session.ts`

If you use refresh tokens with rotation/store, keep session doc referencing user and token metadata.

```
const SessionSchema = new Schema({ userId: { type: Schema.Types.ObjectId,
ref: 'User' }, refreshTokenHash: String, userAgent: String, ip: String,
```

```
  expiresAt: Date }, { timestamps: true });

export default model('Session', SessionSchema);
```

3.6. MonthlyDietReport

Filename: `models/MonthlyDietReport.ts`

Rationale: Aggregate all daily diet plans and progress logs for a given month. Used for user insights and as input for future daily diet generation AI.

```
const MonthlyDietReportSchema = new Schema({
  userId: { type: Schema.Types.ObjectId, ref: 'User', index: true,
required: true },
  year: { type: Number, required: true },
  month: { type: Number, required: true }, // 1-12
  dailyPlans: [{ type: Schema.Types.ObjectId, ref: 'DietPlan' }], // all
diet plans for this month
  dailyProgress: [{ type: Schema.Types.ObjectId, ref: 'ProgressLog' }], //
all progress logs for this month
  adherenceScore: { type: Number, min: 0, max: 100 }, // % of meals logged
vs planned
  avgDailyCalories: Number,
  avgMacros: { protein: Number, carbs: Number, fats: Number },
  totalDaysLogged: Number,
  notes: String,
  generatedAt: { type: Date, default: Date.now }
}, { timestamps: true });

// Compound index for efficient monthly report lookup
MonthlyDietReportSchema.index({ userId: 1, year: 1, month: 1 }, { unique:
true });

export default model('MonthlyDietReport', MonthlyDietReportSchema);
```

DTO:

```
export interface MonthlyDietReportDTO {
  id: string;
  userId: string;
  year: number;
  month: number;
  adherenceScore?: number;
  avgDailyCalories?: number;
  avgMacros?: { protein: number; carbs: number; fats: number };
  totalDaysLogged?: number;
  notes?: string;
  generatedAt: string;
}
```

3.7. MonthlyWorkoutReport

Filename: models/MonthlyWorkoutReport.ts

Rationale: Track workout cycle completion and adherence for a month. Used to inform next workout cycle generation with progressive overload.

```
const MonthlyWorkoutReportSchema = new Schema({
  userId: { type: Schema.Types.ObjectId, ref: 'User', index: true,
required: true },
  year: { type: Number, required: true },
  month: { type: Number, required: true }, // 1-12
  workoutPlanId: { type: Schema.Types.ObjectId, ref: 'WorkoutPlan' }, //
the active workout plan for this month
  completedWorkouts: Number, // total workouts completed
  totalWorkouts: Number, // total workouts scheduled
  adherenceScore: { type: Number, min: 0, max: 100 }, // % completion
  avgDuration: Number, // average workout duration in seconds
  strengthGains: { // optional tracking of strength progression
    exercises: [{ name: String, initialWeight: Number, finalWeight:
Number, improvement: Number }]
  },
  notes: String,
  generatedAt: { type: Date, default: Date.now }
}, { timestamps: true });

// Compound index for efficient monthly report lookup
MonthlyWorkoutReportSchema.index({ userId: 1, year: 1, month: 1 }, {
unique: true });

export default model('MonthlyWorkoutReport', MonthlyWorkoutReportSchema);
```

DTO:

```
export interface MonthlyWorkoutReportDTO {
  id: string;
  userId: string;
  year: number;
  month: number;
  workoutPlanId?: string;
  completedWorkouts?: number;
  totalWorkouts?: number;
  adherenceScore?: number;
  avgDuration?: number;
  strengthGains?: { exercises: Array<{ name: string; initialWeight?:
number; finalWeight?: number; improvement?: number }> };
  notes?: string;
  generatedAt: string;
}
```

---

4. API Contracts — Endpoints, requests & responses

All examples assume `BASE_URL = /api`.

4.1. Auth

POST /api/auth/register

- Body: { name, email, password }
- Response 201: { ok: true, data: { user: UserDTO } }

POST /api/auth/login

- Body: { email, password }
- Behavior: validate creds, set httpOnly `refresh_token` cookie, return access token in body or set `access_token` (if cookie approach prefer returning nothing and rely on `/auth/me`).
- Response 200: { ok: true, data: { user: UserDTO } }

POST /api/auth/refresh

- Body: none (cookie-based) or { refreshToken }
- Response 200: { ok: true, data: { accessToken: string } }

GET /api/auth/me

- Auth: access token
- Response 200: { ok: true, data: { user: UserDTO } }

POST /api/auth/logout

- Clears session & cookies
- Response 200: { ok: true }

4.2. Users

GET /api/users?page=1&limit=20

- Admin only
- Response: { ok: true, data: { items: UserDTO[], total, page, limit } }

GET /api/users/:id

- Admin or owner
- Response: { ok: true, data: UserDTO }

POST /api/users

- Admin creates user
- Body: { email, name, password?, role?, profile? }
- Response 201: { ok: true, data: UserDTO }

PUT /api/users/:id

- Admin or owner; body has editable fields
- Response 200: { ok: true, data: UserDTO }

DELETE /api/users/:id

- Admin only
- Response 204 (no body) or { ok: true }

4.3. Workouts

GET /api/workouts?userId=... (admin may pass userId)

- Response: { ok: true, data: [WorkoutPlanDTO] }

GET /api/workouts/:id

- Response: { ok: true, data: WorkoutPlanDTO }

POST /api/workouts (admin)

- Body: WorkoutPlanDTO (without id)
- Response 201: created

POST /api/workouts/generate

- Body: { userId, preferences, goals }
- Server triggers AI generation job; can be synchronous (blocking) or queued
- Response 202 if queued: { ok: true, data: { jobId } } or 200 with plan

PUT /api/workouts/:id

- Update plan

4.4. Diet

GET /api/diet?userId=...&date=YYYY-MM-DD (admin may pass userId, filter by date)

- Response: { ok: true, data: [DietPlanDTO] }

GET /api/diet/:id

- Response: { ok: true, data: DietPlanDTO }

POST /api/diet (admin or manual creation)

- Body: DietPlanDTO (without id)
- Response 201: created

POST /api/diet/generate

- Body: { userId, date, preferences, previousDayProgressId? }

- Server generates diet plan for specific date based on user profile, preferences, and optional previous day progress
- Response 200: { ok: true, data: DietPlanDTO }

POST /api/diet/generate-daily (NEW - Automated Daily Generation)

- Body: { userId, date } (typically called by cron job)
- Fetches previous day's progress from ProgressLog, user profile, and diet history
- Uses AI/rules engine to generate optimized plan for the specified date
- Response 200: { ok: true, data: DietPlanDTO }

PUT /api/diet/:id

- Update plan

4.5. Workout Cycle Management (NEW)

POST /api/workouts/generate-cycle

- Body: { userId, duration (weeks), startDate, previousReportId? }
- Generates complete workout cycle based on duration, user profile, and optional previous month report
- If previousReportId provided, applies progressive overload based on strength gains
- Response 200: { ok: true, data: WorkoutPlanDTO }

4.6. Monthly Reports (NEW)

GET /api/reports/diet/monthly/:year/:month?userId=...

- Fetches or generates monthly diet report
- Response 200: { ok: true, data: MonthlyDietReportDTO }

GET /api/reports/workout/monthly/:year/:month?userId=...

- Fetches or generates monthly workout report
- Response 200: { ok: true, data: MonthlyWorkoutReportDTO }

POST /api/reports/generate

- Body: { userId, year, month, type: 'diet'|'workout' }
- Manually trigger monthly report generation (typically automated via cron)
- Aggregates all daily plans and progress logs for the specified month
- Response 200: { ok: true, data: MonthlyDietReportDTO | MonthlyWorkoutReportDTO }

4.7. Progress

GET /api/progress?userId=...&from=2025-01-01&to=2025-01-31

- Response: { ok: true, data: { items: ProgressLogDTO[] } }

POST /api/progress/workout

- Body: { userId, date, day, completedExercises, totalExercises, durationSec }

- Response 201: created log

POST /api/progress/meal

- Body: { userId, date, mealName, calories, macros }

GET /api/progress/stats?userId=...&range=30d

- Return aggregated stats (total workouts, avg duration, avg calories, activeDays)

4.8. Analytics

GET /api/analytics/overview

- Admin aggregated platform stats

GET /api/analytics/user/:id?from=&to=

- Return KPIs and small trend arrays (sparklines)
- Example response:

```
{ ok:true, data: { kpis: { activeDays: 12, avgCalories: 2100,
workoutsCompleted: 8 }, trends: { weeklyAdherence: [0.7,0.8,0.6,0.9],
weightTrend: [{date:'2025-10-01',weight:80},...] } } }
```

---

5. Automated Generation & Scheduling

5.1. Daily Diet Generation (Cron Job)

Purpose: Generate personalized daily diet plans automatically based on previous day's adherence and progress.

Implementation:

- **Schedule**: Run daily at midnight (or early morning, e.g., 12:01 AM) in user's timezone
- **Library**: Use `node-cron` or `bull` (job queue) for scheduling
- **Process**:
  1. Fetch all active users who need a diet plan for today
  2. For each user:
     - Fetch yesterday's ProgressLog (diet adherence)
     - Fetch user profile (goals, preferences, restrictions)
     - Fetch last 7-30 days of diet history for patterns
     - Call AI/rules engine to generate optimized plan
     - Save new DietPlan with `generatedFrom: 'auto-daily'`, `date: today`, `previousDayProgressId: yesterdayLogId`
  3. Optionally send push notification or email to user

Example Cron Setup (`services/scheduler.ts`):

```
import cron from 'node-cron';
import { generateDailyDietForAllUsers } from './dietGenerationService';

// Run every day at 12:01 AM
cron.schedule('1 0 * * *', async () => {
  console.log('Starting daily diet generation...');
  await generateDailyDietForAllUsers();
  console.log('Daily diet generation completed.');
});
```

AI/Rules Engine Logic (services/dietGenerationService.ts):

```
async function generateDietForUser(userId: string, date: Date) {
  // 1. Fetch previous day progress
  const yesterday = new Date(date);
  yesterday.setDate(yesterday.getDate() - 1);
  const prevProgress = await ProgressLog.findOne({ userId, date: yesterday
});

  // 2. Fetch user profile
  const user = await User.findById(userId);

  // 3. Calculate adherence score
  const adherenceScore = prevProgress ? calculateAdherence(prevProgress) :
100;

  // 4. Adjust calories/macros based on adherence and goals
  let targetCalories = user.profile.targetCalories || 2000;
  if (adherenceScore < 50) {
    // User struggling - make plan easier/more flexible
    targetCalories += 100;
  } else if (adherenceScore > 90 && user.profile.goals.includes('weight-
loss')) {
    // High adherence - can create slight deficit
    targetCalories -= 50;
  }

  // 5. Generate meal plan (call AI API or use rule-based templates)
  const meals = await generateMeals(targetCalories, user.profile.macros,
user.profile.preferences);

  // 6. Save plan
  const dietPlan = new DietPlan({
    userId,
    date,
    dailyCalories: targetCalories,
    meals,
    generatedFrom: 'auto-daily',
    previousDayProgressId: prevProgress?._id,
    notes: `Generated based on ${adherenceScore}% adherence from previous
day`
```

```
  });

  return await dietPlan.save();
}
```

5.2. Monthly Report Generation (Cron Job)

Purpose: Aggregate daily plans and progress into monthly reports for analytics and future plan generation.

Implementation:

- **Schedule**: Run on 1st of each month at 1:00 AM
- **Process**:
  1. For previous month (e.g., if today is Feb 1, process January)
  2. For each active user:
     - Aggregate all DietPlans for that month
     - Aggregate all ProgressLogs for that month
     - Calculate: adherenceScore, avgDailyCalories, avgMacros, totalDaysLogged
     - Save MonthlyDietReport
     - Aggregate all WorkoutPlans and ProgressLogs
     - Calculate: completedWorkouts, adherenceScore, avgDuration, strengthGains
     - Save MonthlyWorkoutReport

Example Cron Setup:

```
import cron from 'node-cron';
import { generateMonthlyReports } from './reportService';

// Run on 1st of every month at 1:00 AM
cron.schedule('0 1 1 * *', async () => {
  console.log('Starting monthly report generation...');
  const prevMonth = new Date();
  prevMonth.setMonth(prevMonth.getMonth() - 1);
  await generateMonthlyReports(prevMonth.getFullYear(),
prevMonth.getMonth() + 1);
  console.log('Monthly report generation completed.');
});
```

5.3. Workout Cycle Completion & Next Cycle Trigger

Purpose: When a workout cycle ends, automatically trigger generation of next cycle using previous month's report.

Implementation:

- **Trigger**: Check daily for WorkoutPlans where `endDate` is yesterday and `status: 'active'`

- **Process**:
  1. Mark WorkoutPlan as `status: 'completed'`
  2. Fetch most recent MonthlyWorkoutReport for that user
  3. Call `/api/workouts/generate-cycle` with previousReportId
  4. Apply progressive overload based on strength gains from report
  5. Notify user that new cycle is ready

Example Check (runs daily):

```
cron.schedule('0 2 * * *', async () => {
  const yesterday = new Date();
  yesterday.setDate(yesterday.getDate() - 1);
  yesterday.setHours(0, 0, 0, 0);

  // Find plans that ended yesterday
  const completedPlans = await WorkoutPlan.find({
    status: 'active',
    endDate: { $gte: yesterday, $lt: new Date(yesterday.getTime() + 24 *
60 * 60 * 1000) }
  });

  for (const plan of completedPlans) {
    plan.status = 'completed';
    await plan.save();

    // Get last month report
    const now = new Date();
    const report = await MonthlyWorkoutReport.findOne({
      userId: plan.userId,
      year: now.getFullYear(),
      month: now.getMonth()
    });

    // Generate next cycle
    await generateNextWorkoutCycle(plan.userId, plan.duration,
report?._id);
  }
});
```

5.4. Timezone Considerations

Challenge: Users in different timezones need plans generated at their local midnight.

Solutions:

1. **Store user timezone** in User.profile.timezone (e.g., 'America/New_York')
2. **Batch processing by timezone**: Group users by timezone, run cron jobs staggered
3. **Job Queue (Recommended)**: Use Bull/BullMQ with delayed jobs
   - At server's midnight, enqueue jobs for each timezone
   - Each job processes users in that timezone when their midnight arrives

Example with Bull:

```
import Queue from 'bull';

const dietQueue = new Queue('daily-diet-generation');

// Enqueue jobs for all timezones
cron.schedule('0 0 * * *', async () => {
  const timezones = ['America/New_York', 'America/Los_Angeles',
'Europe/London', 'Asia/Tokyo'];
  for (const tz of timezones) {
    const usersInTz = await User.find({ 'profile.timezone': tz });
    for (const user of usersInTz) {
      const localMidnight = getNextMidnightInTimezone(tz);
      dietQueue.add({ userId: user._id, date: localMidnight }, { delay:
localMidnight.getTime() - Date.now() });
    }
  }
});

// Process jobs
dietQueue.process(async (job) => {
  await generateDietForUser(job.data.userId, job.data.date);
});
```

---

6. Frontend Integration (Practical)

This section explains how the existing Next.js frontend should call the API and what hooks/types to use.

6.1. Base API client

File: `gym-app/lib/api.ts` (update)

Example (fetch wrapper):

```
const API_BASE = process.env.NEXT_PUBLIC_API_BASE_URL ||
'http://localhost:4000/api';

async function apiFetch<T>(path: string, opts: RequestInit = {}) {
  const url = `${API_BASE}${path}`;
  const res = await fetch(url, { credentials: 'include', headers: {
'Content-Type': 'application/json' }, ...opts });
  const body = await res.json().catch(() => null);

  if (!res.ok) {
    throw body || { message: res.statusText };
  }
  return body as { ok: boolean; data: T };
```

```
}

export default apiFetch;
```

Notes:

- `credentials: 'include'` is required when using httpOnly refresh cookie
- Optionally implement automatic refresh on 401: intercept response, call `/auth/refresh`, retry original

6.2. Hooks (examples)

Use SWR or React Query. Example with SWR:

`hooks/useUser.ts`

```
import useSWR from 'swr';
import apiFetch from '@/lib/api';

export function useMe() {
  const { data, error, mutate } = useSWR('/auth/me', (url) =>
apiFetch('/auth/me').then(r => r.data));
  return { user: data, isLoading: !error && !data, isError: error, mutate
};
}
```

`hooks/useWorkouts.ts`

```
export function useWorkouts(userId?: string) {
  const key = userId ? `/workouts?userId=${userId}` : '/workouts';
  const { data } = useSWR(key, () => apiFetch(key).then(r => r.data));
  return data as WorkoutPlanDTO[] | undefined;
}
```

`hooks/useDailyDiet.ts` (NEW)

```
export function useDailyDiet(userId: string, date: string) {
  const key = `/diet?userId=${userId}&date=${date}`;
  const { data, error, mutate } = useSWR(key, () => apiFetch(key).then(r
=> r.data));
  return { dietPlan: data?.[0], isLoading: !error && !data, isError:
error, mutate };
}
```

`hooks/useMonthlyDietReport.ts` (NEW)

```
export function useMonthlyDietReport(userId: string, year: number, month:
number) {
  const key = `/reports/diet/monthly/${year}/${month}?userId=${userId}`;
  const { data, error } = useSWR(key, () => apiFetch(key).then(r =>
r.data));
  return { report: data, isLoading: !error && !data, isError: error };
}
```

`hooks/useMonthlyWorkoutReport.ts` (NEW)

```
export function useMonthlyWorkoutReport(userId: string, year: number,
month: number) {
  const key = `/reports/workout/monthly/${year}/${month}?
userId=${userId}`;
  const { data, error } = useSWR(key, () => apiFetch(key).then(r =>
r.data));
  return { report: data, isLoading: !error && !data, isError: error };
}
```

6.3. Auth flow details

Recommended cookie strategy flow:

1. Login form posts to `/api/auth/login` -> server sets `refresh_token` cookie (httpOnly, secure in prod).
2. Server may return `/auth/me` or access token in body. Prefer simply returning `user` and letting frontend fetch `/auth/me` to confirm.
3. For protected API calls, include `credentials:'include'`. If access token usage required, send as `Authorization: Bearer <token>`.
4. Refresh on 401 by calling `/auth/refresh` endpoint which uses `refresh_token` cookie to issue a new access token.

Frontend guard examples:

- On pages that must be protected, call `useMe()` and if not logged in redirect to `/login`.
- Alternatively use Next middleware for server-side redirects.

6.4. Type safety

Add shared TypeScript types under `gym-app/types` to mirror backend DTOs. Example:

- `types/user.ts`, `types/workout.ts`, `types/diet.ts`, `types/progress.ts`
- **`types/reports.ts` (NEW)** - for monthly report DTOs

Keep the type shapes stable and version when you change the backend (v1/v2 paths).

New types to add (`types/reports.ts`):

```
export interface MonthlyDietReportDTO {
  id: string;
  userId: string;
  year: number;
  month: number;
  adherenceScore?: number;
  avgDailyCalories?: number;
  avgMacros?: { protein: number; carbs: number; fats: number };
  totalDaysLogged?: number;
  notes?: string;
  generatedAt: string;
}

export interface MonthlyWorkoutReportDTO {
  id: string;
  userId: string;
  year: number;
  month: number;
  workoutPlanId?: string;
  completedWorkouts?: number;
  totalWorkouts?: number;
  adherenceScore?: number;
  avgDuration?: number;
  strengthGains?: {
    exercises: Array<{
      name: string;
      initialWeight?: number;
      finalWeight?: number;
      improvement?: number;
    }>;
  };
  notes?: string;
  generatedAt: string;
}
```

Update existing types:

- `types/diet.ts`: Add `date`, `generatedFrom`, `previousDayProgressId`, `notes` to `DietPlanDTO`
- `types/workout.ts`: Add `duration`, `startDate`, `endDate`, `status`, `generatedFrom` to `WorkoutPlanDTO`

6.5. Error handling & UX

- Show friendly messages when API returns `ok: false`
- Add skeleton loaders while SWR/React Query fetches
- Use optimistic updates carefully (e.g., marking exercise complete) and rollback on error

6.6. Example: Mark exercise complete

Frontend sends POST to `/api/progress/workout` with body:

```
{ "userId": "...", "date": "2025-11-01", "day": "monday",
"completedExercises": 6, "totalExercises": 8, "durationSec": 1800 }
```

Server stores or upserts a ProgressLog entry. Frontend `mutate()` the SWR key for `/progress?userId=...&from=...` to refresh.

7. Analytics: Aggregation Examples

7.1. Active days last 30 days (aggregation)

```
// ProgressLog collection
db.progresslogs.aggregate([
  { $match: { userId: ObjectId(USER_ID), date: { $gte: ISODate(from) } }
},
  { $group: { _id: { $dateToString: { format: "%Y-%m-%d", date: "$date" }
}, workouts: { $sum: { $cond: [ { $ifNull: ["$workout", false ] }, 1, 0 ]
} } } },
  { $sort: { _id: 1 } }
]);
```

7.2. Weekly adherence (completedExercises / totalExercises)

```
db.progresslogs.aggregate([
  { $match: { userId: ObjectId(USER_ID), date: { $gte: ISODate(from) } }
},
  { $project: { date: 1, adherence: { $cond: [ { $eq:
["$workout.totalExercises", 0] }, 0, { $divide:
["$workout.completedExercises", "$workout.totalExercises"] } ] } } },
  { $group: { _id: { $isoWeek: "$date" }, avgAdherence: { $avg:
"$adherence" } } },
  { $sort: { _id: 1 } }
]);
```

Cache this result for heavy dashboards (Redis TTL 30m).

7.3. Monthly Diet Report Aggregation (NEW)

```
// Aggregate diet adherence for a specific month
db.progresslogs.aggregate([
  {
    $match: {
      userId: ObjectId(USER_ID),
      date: { $gte: ISODate("2025-01-01"), $lt: ISODate("2025-02-01") }
    }
  },
```

```
  {
    $group: {
      _id: null,
      totalDays: { $sum: 1 },
      avgCalories: { $avg: { $sum: "$meals.calories" } },
      totalMealsLogged: { $sum: { $size: "$meals" } }
    }
  },
  {
    $lookup: {
      from: "dietplans",
      let: { uid: "$userId" },
      pipeline: [
        { $match: {
          $expr: { $eq: ["$userId", "$$uid"] },
          date: { $gte: ISODate("2025-01-01"), $lt: ISODate("2025-02-01")
}
        }},
        { $count: "totalPlannedDays" }
      ],
      as: "plannedDays"
    }
  },
  {
    $project: {
      totalDaysLogged: "$totalDays",
      avgDailyCalories: "$avgCalories",
      adherenceScore: {
        $multiply: [
          { $divide: ["$totalDays", { $arrayElemAt:
["$plannedDays.totalPlannedDays", 0] }] },
          100
        ]
      }
    }
  }
]);
```

7.4. Monthly Workout Report Aggregation (NEW)

```
// Aggregate workout completion for a specific month
db.progresslogs.aggregate([
  {
    $match: {
      userId: ObjectId(USER_ID),
      date: { $gte: ISODate("2025-01-01"), $lt: ISODate("2025-02-01") },
      "workout": { $exists: true }
    }
  },
  {
    $group: {
```

```
      _id: null,
      completedWorkouts: { $sum: 1 },
      totalCompletedExercises: { $sum: "$workout.completedExercises" },
      totalPlannedExercises: { $sum: "$workout.totalExercises" },
      avgDuration: { $avg: "$workout.durationSec" }
    }
  },
  {
    $project: {
      completedWorkouts: 1,
      avgDuration: 1,
      adherenceScore: {
        $multiply: [
          { $divide: ["$totalCompletedExercises",
"$totalPlannedExercises"] },
          100
        ]
      }
    }
  }
]);
```

---

8. Indexes & Performance

Required Indexes:

- `User.email` - unique index for auth lookups
- `WorkoutPlan.userId` - for user's workout queries
- `WorkoutPlan.status` - for filtering active/completed plans
- **`DietPlan.userId + DietPlan.date`** - compound index for daily diet lookups (NEW)
- `ProgressLog.userId + ProgressLog.date` - compound index for progress timelines
- **`MonthlyDietReport.userId + MonthlyDietReport.year + MonthlyDietReport.month`** - unique compound index (NEW)
- **`MonthlyWorkoutReport.userId + MonthlyWorkoutReport.year + MonthlyWorkoutReport.month`** - unique compound index (NEW)
- `Session.userId` - for session management
- `Session.expiresAt` - TTL index for automatic session cleanup

Performance Tips:

- Use `.lean()` on read-only queries to reduce overhead
- Cache monthly reports in Redis (TTL 1 hour) since they don't change often
- For daily diet generation, batch users by timezone to avoid timezone conversion on every query
- Consider read replicas for heavy analytics queries
- Use aggregation pipeline `$facet` to get multiple stats in one query

---

9. Edge Cases & Validation

- Handle daylight/date zones: store `date` as UTC midnight for daily logs.
- Partial updates: `PATCH` endpoints or `PUT` with full body
- Concurrent updates: optimistic concurrency with `updatedAt` or Mongo's versioning
- Large plans: paginate or lazy-load exercises

Validation

- Use Zod to validate incoming shape and return consistent error envelope

---

10. Migration & Versioning

- Start with v1 routes: `/api/v1/...` (optional)
- Migrations: use `migrate-mongo` or custom scripts stored in `db/migrations`
- When changing JSON contracts, bump API minor version and add transformation scripts for old DB rows

---

11. Testing & Seeding

- Provide `scripts/seed.ts` to create demo users/plan data
- Tests:
  - Unit: services and utils
  - Integration: routes via supertest with test DB (mongodb-memory-server)

---

12. Implementation Checklist (practical next steps)

Phase 1: Core Setup

1. Scaffold Express app + env validation + DB connect. (server, app, config/db)
2. Add `User` model + auth routes + session model. Implement register/login/refresh/me/logout.
3. Create `WorkoutPlan` & `DietPlan` models and CRUD routes.
4. Add `ProgressLog` model with POST endpoints for logging workouts and meals.
5. Implement basic analytics aggregation queries.

Phase 2: Monthly Reports & Automated Generation (NEW)

6. Create `MonthlyDietReport` and `MonthlyWorkoutReport` models.
7. Update `DietPlan` schema to include `date`, `generatedFrom`, and `previousDayProgressId` fields.
8. Update `WorkoutPlan` schema to include `duration`, `startDate`, `endDate`, `status`, and `generatedFrom` fields.
9. Implement monthly report aggregation endpoints:
   - GET `/reports/diet/monthly/:year/:month`
   - GET `/reports/workout/monthly/:year/:month`
   - POST `/reports/generate`
10. Build aggregation service to compute monthly statistics from daily data.

Phase 3: Daily Diet Generation System

11. Set up OpenRouter and RapidAPI integration:
    - Get API keys (both free)
    - Create config files for OpenRouter client
    - Set up axios instances for RapidAPI
    - Implement caching layer (node-cache) for API responses
12. Implement `dietGenerationService.ts`:
    - Build AI prompt generator with user context
    - Implement OpenRouter chat completion
    - Add RapidAPI nutrition data enrichment
    - Build rule-based fallback system
13. Implement POST `/diet/generate-daily` endpoint:
    - Fetch previous day's ProgressLog
    - Calculate adherence score
    - Call dietGenerationService
    - Save generated plan with proper metadata
14. Set up cron scheduler (node-cron):
    - Daily diet generation at midnight (timezone-aware)
    - Monthly report generation on 1st of month
    - Workout cycle completion check and next cycle trigger
15. Add timezone support to User model and implement timezone-based job scheduling.

Phase 4: Workout Cycle System

16. Implement `workoutGenerationService.ts`:
    - Build workout prompt with periodization logic
    - Use OpenRouter smart model (Llama 3.1) for workout generation
    - Implement RapidAPI ExerciseDB enrichment (animations, instructions)
    - Add progressive overload calculator based on previous report
17. Implement POST `/workouts/generate-cycle` endpoint:
    - Accept duration parameter (weeks)
    - Fetch previous MonthlyWorkoutReport if available
    - Call workoutGenerationService
    - Calculate start/end dates and save plan
18. Add workout cycle status tracking and auto-completion logic.

Phase 5: Frontend Integration

19. Frontend: implement `lib/api.ts`, replace localStorage mocks.
20. Wire up hooks: `useAuth`, `useWorkouts`, `useDietPlan`, `useProgress`, `useDailyDiet`, `useMonthlyDietReport`, `useMonthlyWorkoutReport`.
21. Add JWT cookie flow and test full login → protected endpoints flow.
22. Build monthly report dashboard pages for users:
    - Monthly diet adherence charts
    - Monthly workout completion stats
    - Progress trends and insights
23. Add UI indicators for auto-generated plans:

    - Badge showing "□ AI Generated from yesterday's progress"
    - Show adherence score that influenced today's plan
    - Display adjustments made (e.g., "+100 calories - easier plan")
24. Implement loading states and fallback UI for generation failures.

Phase 6: Testing & Deployment

25. Write unit tests for generation services:
    - Test prompt building logic
    - Test calorie calculation formulas
    - Test adherence adjustments
    - Mock OpenRouter and RapidAPI responses
26. Integration tests for cron jobs and generation logic:
    - Test daily diet generation for multiple users
    - Test monthly report aggregation
    - Test workout cycle transitions
27. Add seed scripts:
    - Sample users with different goals
    - Sample progress logs for testing adherence
    - Sample monthly reports for testing progressive overload
28. Load testing:
    - Test daily generation at scale (1000+ users)
    - Test API rate limiting and caching
    - Test fallback systems under load
29. Deploy with proper environment-based scheduling:
    - Set up OpenRouter and RapidAPI production keys
    - Configure cron jobs for production timezone
    - Set up monitoring and alerts for generation failures
    - Configure rate limiting and caching in production

---

13. Example: Minimal API Endpoint Implementation (pseudo)

`POST /api/progress/workout` controller pseudo:

```
// controller
export async function logWorkout(req, res) {
  const payload = req.body; // validated via zod
  await ProgressLogModel.updateOne({ userId: payload.userId, date:
startOfDay(payload.date) }, { $set: { workout: payload } }, { upsert: true
});
  return res.status(201).json({ ok: true });
}
```

Frontend call (example):

```
await apiFetch('/progress/workout', { method: 'POST', body:
JSON.stringify(payload) });
mutate(`/progress?userId=${userId}&from=${from}&to=${to}`);
```

---

14. Follow-ups / Decisions Made □

- □ **Cookie vs Bearer token**: Cookie-based with httpOnly for Next.js
- □ **AI provider**: OpenRouter with free models (Gemini Flash, Llama 3.1, Mistral)
- □ **Data enrichment**: RapidAPI (Nutritionix for nutrition, ExerciseDB for workouts)
- □ **Generation approach**: Synchronous for MVP, with multi-tier fallback system
- □ **Caching strategy**: node-cache for nutrition/exercise data (24hr TTL), Redis for production
- □ **Analytics cache**: 1 hour TTL for monthly reports, real-time for daily stats
- □ **Timezone handling**: Simple cron with batched processing (MVP), Bull queue for scale
- □ **Adherence scoring**: `(logged meals / planned meals) * 100`, simple and effective
- □ **Progressive overload**: 5-10% weight increase or +1-2 reps based on previous cycle
- □ **Cost structure**: 100% free tier usage - OpenRouter free models + RapidAPI free tier

Remaining Decisions:

- □ **Plan versioning**: Track evolution? (Nice-to-have, not MVP)
- □ **User feedback**: Like/dislike meals? (Phase 2 - ML training data)
- □ **Grocery lists**: Auto-generate shopping lists from meal plans? (Phase 3)
- □ **Meal prep**: Batch cooking suggestions? (Phase 3)
- □ **Social features**: Share workouts/meals with friends? (Phase 4)

---

17. Environment Variables & Setup

Backend `.env` file:

```
# Server
PORT=4000
NODE_ENV=development

# Database
MONGODB_URI=mongodb://localhost:27017/fitflow
MONGODB_URI_PROD=mongodb+srv://user:pass@cluster.mongodb.net/fitflow

# Auth
JWT_SECRET=your_jwt_secret_key_min_32_chars
JWT_REFRESH_SECRET=your_refresh_secret_key_min_32_chars
JWT_EXPIRES_IN=15m
JWT_REFRESH_EXPIRES_IN=7d

# AI Generation (OpenRouter)
OPENROUTER_API_KEY=your_openrouter_api_key
```

```
APP_URL=http://localhost:3000

# Data Enrichment (RapidAPI)
RAPIDAPI_KEY=your_rapidapi_key

# Redis (optional, for caching)
REDIS_URL=redis://localhost:6379

# CORS
CORS_ORIGIN=http://localhost:3000

# Rate Limiting
RATE_LIMIT_WINDOW_MS=900000
RATE_LIMIT_MAX_REQUESTS=100
```

Getting API Keys:

1. **OpenRouter**:
   - Visit https://openrouter.ai/
   - Sign up (free)
   - Get API key from dashboard
   - Free models available: Llama 3.1, Gemini Flash, Mistral
2. **RapidAPI**:
   - Visit https://rapidapi.com/
   - Sign up (free)
   - Subscribe to APIs:
     - ExerciseDB: https://rapidapi.com/justin-WFnsXH_t6/api/exercisedb
     - Nutritionix: https://rapidapi.com/msilverman/api/nutritionix-nutrition-database
   - Copy API key from dashboard

---

18. Appendices

18.1. Required NPM Packages

Core Dependencies:

```
{
  "dependencies": {
    "express": "^4.18.2",
    "mongoose": "^8.0.0",
    "bcrypt": "^5.1.1",
    "jsonwebtoken": "^9.0.2",
    "zod": "^3.22.4",
    "dotenv": "^16.3.1",
    "cors": "^2.8.5",
    "helmet": "^7.1.0",
    "express-rate-limit": "^7.1.5",
```

```
    "openai": "^4.20.0",
    "axios": "^1.6.2",
    "node-cron": "^3.0.3",
    "node-cache": "^5.1.2",
    "winston": "^3.11.0",
    "morgan": "^1.10.0"
  },
  "devDependencies": {
    "typescript": "^5.3.3",
    "@types/node": "^20.10.4",
    "@types/express": "^4.17.21",
    "@types/bcrypt": "^5.0.2",
    "@types/jsonwebtoken": "^9.0.5",
    "@types/cors": "^2.8.17",
    "@types/morgan": "^1.9.9",
    "@types/node-cron": "^3.0.11",
    "ts-node-dev": "^2.0.0",
    "jest": "^29.7.0",
    "supertest": "^6.3.3",
    "mongodb-memory-server": "^9.1.3",
    "eslint": "^8.55.0",
    "prettier": "^3.1.0"
  }
}
```

Optional (Production):

```
{
  "dependencies": {
    "bull": "^4.12.0",
    "bullmq": "^5.0.0",
    "redis": "^4.6.11",
    "ioredis": "^5.3.2",
    "swagger-ui-express": "^5.0.0",
    "express-validator": "^7.0.1"
  }
}
```

18.2. Installation Commands

```
# Create backend folder
mkdir fitflow-api
cd fitflow-api

# Initialize project
npm init -y

# Install dependencies
npm install express mongoose bcrypt jsonwebtoken zod dotenv cors helmet
```

```
express-rate-limit openai axios node-cron node-cache winston morgan

# Install dev dependencies
npm install -D typescript @types/node @types/express @types/bcrypt
@types/jsonwebtoken @types/cors @types/morgan @types/node-cron ts-node-dev
jest supertest mongodb-memory-server

# Initialize TypeScript
npx tsc --init

# Create folder structure
mkdir -p
src/{config,models,routes,controllers,services,middleware,utils,tests}
touch src/server.ts src/app.ts .env .env.example
```

18.3. Quick Start Script

File: `scripts/setup.sh`

```
#!/bin/bash

echo "🚀 Setting up FitFlow Backend..."

# Install dependencies
npm install

# Copy env template
cp .env.example .env

echo "📝 Please update .env with your API keys:"
echo "  - OPENROUTER_API_KEY"
echo "  - RAPIDAPI_KEY"
echo "  - JWT_SECRET"
echo ""

# Start MongoDB (if using Docker)
docker run -d -p 27017:27017 --name fitflow-mongo mongo:latest

echo "✅ Setup complete!"
echo "Run 'npm run dev' to start the server"
```

18.4. Recommended Dev Tooling

- **IDE**: VS Code with extensions: ESLint, Prettier, MongoDB for VS Code
- **Testing**: Jest + Supertest + mongodb-memory-server
- **Debugging**: VS Code debugger with ts-node
- **API Testing**: Postman or Thunder Client (VS Code extension)
- **Monitoring**: Winston (logging) + Morgan (HTTP logs)

16. Plan Generation Strategy (AI-Powered Hybrid)

16.1. Architecture Overview

Hybrid Approach: OpenRouter (Free AI Models) + RapidAPI (Nutrition Data) + Rule-Based Templates

```
User Request → Rules Engine (Calculate Base) → AI Generation (OpenRouter)
→ Data Enrichment (RapidAPI) → Final Plan
```

Benefits:

- **Cost-effective**: OpenRouter provides free access to models like Llama, Mistral, Gemini
- **Rich data**: RapidAPI for nutrition facts, exercise databases, meal ideas
- **Fast & reliable**: Rule-based fallback if APIs fail
- **Scalable**: Queue-based generation for high load

16.2. OpenRouter Integration (Free AI Models)

Setup:

```
npm install openai  # OpenRouter uses OpenAI SDK format
```

Configuration (`config/openrouter.ts`):

```
import OpenAI from 'openai';

export const openrouter = new OpenAI({
  baseURL: "https://openrouter.ai/api/v1",
  apiKey: process.env.OPENROUTER_API_KEY,
  defaultHeaders: {
    "HTTP-Referer": process.env.APP_URL || "http://localhost:3000",
    "X-Title": "FitFlow - AI Fitness Platform",
  }
});

// Free models available on OpenRouter
export const FREE_MODELS = {
  FAST: "google/gemini-2.0-flash-exp:free",        // Fast responses
  SMART: "meta-llama/llama-3.1-8b-instruct:free",  // Better reasoning
  BALANCED: "mistralai/mistral-7b-instruct:free"   // Good balance
};
```

Environment Variables (`.env`):

```
OPENROUTER_API_KEY=your_openrouter_api_key
RAPIDAPI_KEY=your_rapidapi_key
APP_URL=http://localhost:3000
```

16.3. Diet Plan Generation Service

File: services/dietGenerationService.ts

```
import { openrouter, FREE_MODELS } from '../config/openrouter';
import axios from 'axios';

interface DietGenerationInput {
  userId: string;
  date: Date;
  user: {
    age: number;
    weight: number;
    height: number;
    gender: string;
    goals: string[];
    activityLevel: string;
    preferences?: string[];
    restrictions?: string[];
  };
  previousDayProgress?: {
    adherenceScore: number;
    totalCalories: number;
    missedMeals: string[];
  };
}

export async function generateDailyDietPlan(input: DietGenerationInput) {
  try {
    // 1. Calculate base calories using rules
    const baseCalories = calculateTargetCalories(input.user,
input.previousDayProgress);
    const macros = calculateMacros(baseCalories, input.user.goals);

    // 2. Generate meal plan using OpenRouter AI
    const prompt = buildDietPrompt(input.user, baseCalories, macros,
input.previousDayProgress);

    const aiResponse = await openrouter.chat.completions.create({
      model: FREE_MODELS.FAST,
      messages: [
        {
          role: "system",
          content: "You are a professional nutritionist. Generate
personalized meal plans in JSON format."
        },
        {
```

```
          role: "user",
          content: prompt
        }
      ],
      response_format: { type: "json_object" },
      temperature: 0.7,
      max_tokens: 2000
    });

    const aiMeals = JSON.parse(aiResponse.choices[0].message.content);

    // 3. Enrich meals with nutrition data from RapidAPI
    const enrichedMeals = await
enrichMealsWithNutritionData(aiMeals.meals);

    // 4. Save to database
    const dietPlan = new DietPlan({
      userId: input.userId,
      date: input.date,
      name: aiMeals.name || 'Daily Diet Plan',
      dailyCalories: baseCalories,
      macros,
      meals: enrichedMeals,
      generatedFrom: 'ai',
      previousDayProgressId: input.previousDayProgress?.id,
      notes: aiMeals.notes || ''
    });

    return await dietPlan.save();

  } catch (error) {
    console.error('AI generation failed, using rule-based fallback:',
error);
    // Fallback to rule-based template selection
    return generateDietPlanRuleBased(input);
  }
}

function buildDietPrompt(user, calories, macros, previousProgress?) {
  return `
Generate a personalized daily meal plan with these requirements:

**User Profile:**
- Age: ${user.age}, Weight: ${user.weight}kg, Height: ${user.height}cm,
Gender: ${user.gender}
- Goals: ${user.goals.join(', ')}
- Activity Level: ${user.activityLevel}
- Dietary Preferences: ${user.preferences?.join(', ') || 'None'}
- Restrictions: ${user.restrictions?.join(', ') || 'None'}

**Targets:**
- Daily Calories: ${calories} kcal
- Macros: ${macros.protein}g protein, ${macros.carbs}g carbs,
${macros.fats}g fats
```

```
${previousProgress ? `
**Previous Day Performance:**
- Adherence: ${previousProgress.adherenceScore}%
- Calories consumed: ${previousProgress.totalCalories} kcal
- Missed meals: ${previousProgress.missedMeals.join(', ')}
${previousProgress.adherenceScore < 50 ? '⚠ User struggled yesterday -
make plan simpler and more flexible' : '□ User doing well - can maintain
or increase challenge'}
` : ''}

**Output Format (JSON):**
{
  "name": "Plan name",
  "meals": [
    {
      "name": "Breakfast",
      "time": "08:00",
      "calories": 400,
      "foods": [
        {
          "name": "Oatmeal",
          "portion": "1 cup cooked",
          "calories": 150,
          "macros": { "p": 5, "c": 27, "f": 3 }
        }
      ]
    }
  ],
  "notes": "Tips for success"
}

Generate 4-5 meals (breakfast, snack, lunch, snack, dinner) that are
realistic, affordable, and easy to prepare.
`;
}

async function enrichMealsWithNutritionData(meals) {
  // Use RapidAPI Nutritionix or Edamam for accurate nutrition data
  const enrichedMeals = await Promise.all(meals.map(async (meal) => {
    const enrichedFoods = await Promise.all(meal.foods.map(async (food) =>
{
      try {
        const nutritionData = await fetchNutritionData(food.name,
food.portion);
        return {
          ...food,
          calories: nutritionData.calories || food.calories,
          macros: nutritionData.macros || food.macros
        };
      } catch (error) {
        // Keep AI-generated data if API fails
        return food;
      }
```

```
    }));

    return {
      ...meal,
      foods: enrichedFoods,
      calories: enrichedFoods.reduce((sum, f) => sum + (f.calories || 0),
0)
    };
  }));

  return enrichedMeals;
}

async function fetchNutritionData(foodName: string, portion: string) {
  try {
    const response = await axios.get('https://nutritionix-
api.p.rapidapi.com/v1_1/search', {
      params: { query: foodName },
      headers: {
        'X-RapidAPI-Key': process.env.RAPIDAPI_KEY,
        'X-RapidAPI-Host': 'nutritionix-api.p.rapidapi.com'
      }
    });

    const foodData = response.data.hits[0]?.fields;
    return {
      calories: foodData?.nf_calories || 0,
      macros: {
        p: foodData?.nf_protein || 0,
        c: foodData?.nf_total_carbohydrate || 0,
        f: foodData?.nf_total_fat || 0
      }
    };
  } catch (error) {
    console.error('RapidAPI nutrition fetch failed:', error);
    return null;
  }
}

// Rule-based fallback
function generateDietPlanRuleBased(input: DietGenerationInput) {
  // Use pre-built templates
  const mealTemplates = getMealTemplates(input.user.goals,
input.user.preferences);
  const selectedMeals = selectMealsToMatchMacros(mealTemplates,
input.user);

  return new DietPlan({
    userId: input.userId,
    date: input.date,
    meals: selectedMeals,
    generatedFrom: 'manual',
    // ... rest of plan
  }).save();
```

```
}

function calculateTargetCalories(user, previousProgress?) {
  // Calculate BMR (Basal Metabolic Rate)
  let bmr;
  if (user.gender === 'male') {
    bmr = 10 * user.weight + 6.25 * user.height - 5 * user.age + 5;
  } else {
    bmr = 10 * user.weight + 6.25 * user.height - 5 * user.age - 161;
  }

  // Apply activity multiplier
  const activityMultipliers = {
    sedentary: 1.2,
    light: 1.375,
    moderate: 1.55,
    active: 1.725,
    veryActive: 1.9
  };
  let tdee = bmr * (activityMultipliers[user.activityLevel] || 1.2);

  // Adjust for goals
  if (user.goals.includes('weight-loss')) tdee -= 500;
  if (user.goals.includes('muscle-gain')) tdee += 300;

  // Adjust based on previous day adherence
  if (previousProgress) {
    if (previousProgress.adherenceScore < 50) {
      tdee += 100; // Make it easier
    } else if (previousProgress.adherenceScore > 90 &&
user.goals.includes('weight-loss')) {
      tdee -= 50; // Can be more aggressive
    }
  }

  return Math.round(tdee);
}

function calculateMacros(calories: number, goals: string[]) {
  let proteinPercent = 0.30;
  let carbsPercent = 0.40;
  let fatsPercent = 0.30;

  if (goals.includes('muscle-gain')) {
    proteinPercent = 0.35;
    carbsPercent = 0.45;
    fatsPercent = 0.20;
  } else if (goals.includes('weight-loss')) {
    proteinPercent = 0.40;
    carbsPercent = 0.30;
    fatsPercent = 0.30;
  }

  return {
```

```
    protein: Math.round((calories * proteinPercent) / 4),
    carbs: Math.round((calories * carbsPercent) / 4),
    fats: Math.round((calories * fatsPercent) / 9)
  };
}
```

16.4. Workout Plan Generation Service

File: services/workoutGenerationService.ts

```
import { openrouter, FREE_MODELS } from '../config/openrouter';
import axios from 'axios';

interface WorkoutGenerationInput {
  userId: string;
  duration: number; // weeks
  startDate: Date;
  user: {
    age: number;
    goals: string[];
    experience: string; // beginner, intermediate, advanced
    availableDays: number;
    equipment?: string[];
  };
  previousReport?: {
    adherenceScore: number;
    strengthGains: Array<{
      name: string;
      initialWeight: number;
      finalWeight: number;
    }>;
  };
}

export async function generateWorkoutCycle(input: WorkoutGenerationInput)
{
  try {
    // 1. Build prompt with progressive overload data
    const prompt = buildWorkoutPrompt(input);

    // 2. Generate workout plan using OpenRouter
    const aiResponse = await openrouter.chat.completions.create({
      model: FREE_MODELS.SMART, // Use smarter model for workout logic
      messages: [
        {
          role: "system",
          content: "You are an expert strength coach and personal trainer.
Generate scientifically-backed workout programs in JSON format with proper
periodization and progressive overload."
        },
        {
```

```
            role: "user",
            content: prompt
          }
        ],
        response_format: { type: "json_object" },
        temperature: 0.5,
        max_tokens: 3000
      });

      const aiWorkout = JSON.parse(aiResponse.choices[0].message.content);

      // 3. Enrich exercises with data from RapidAPI ExerciseDB
      const enrichedDays = await enrichExercisesWithData(aiWorkout.days);

      // 4. Calculate end date
      const endDate = new Date(input.startDate);
      endDate.setDate(endDate.getDate() + input.duration * 7);

      // 5. Save to database
      const workoutPlan = new WorkoutPlan({
        userId: input.userId,
        name: aiWorkout.name,
        duration: input.duration,
        startDate: input.startDate,
        endDate: endDate,
        status: 'active',
        days: enrichedDays,
        generatedFrom: input.previousReport ? input.previousReport._id :
undefined
      });

      return await workoutPlan.save();

    } catch (error) {
      console.error('AI workout generation failed, using template:', error);
      return generateWorkoutPlanRuleBased(input);
    }
}

function buildWorkoutPrompt(input: WorkoutGenerationInput) {
  return `
Generate a ${input.duration}-week workout program with these requirements:

**User Profile:**
- Age: ${input.user.age}
- Goals: ${input.user.goals.join(', ')}
- Experience Level: ${input.user.experience}
- Available Days: ${input.user.availableDays} days per week
- Equipment: ${input.user.equipment?.join(', ') || 'Full gym access'}

${input.previousReport ? `
**Previous Cycle Performance:**
- Adherence: ${input.previousReport.adherenceScore}%
- Strength Gains:
```

```
${input.previousReport.strengthGains.map(e =>
  `  • ${e.name}: ${e.initialWeight}kg → ${e.finalWeight}kg
(+${((e.finalWeight - e.initialWeight) / e.initialWeight *
100).toFixed(1)}%)`
).join('\n')}

⚡ Apply progressive overload: Increase weights by 5-10% or reps by 1-2
based on previous performance.
` : ''}

**Output Format (JSON):**
{
  "name": "Program name",
  "days": [
    {
      "day": "Monday - Push",
      "exercises": [
        {
          "name": "Bench Press",
          "sets": 4,
          "reps": "8-10",
          "rest": 120,
          "notes": "Focus on controlled eccentric"
        }
      ]
    }
  ]
}

Create a periodized program with:
- Progressive overload built in
- Proper exercise selection for goals
- Deload week if duration > 4 weeks
- Compound movements prioritized
- Appropriate volume per muscle group
`;
}

async function enrichExercisesWithData(days) {
  const enrichedDays = await Promise.all(days.map(async (day) => {
    const enrichedExercises = await Promise.all(day.exercises.map(async
(exercise) => {
      try {
        const exerciseData = await fetchExerciseData(exercise.name);
        return {
          ...exercise,
          targetMuscles: exerciseData?.targetMuscles || [],
          equipment: exerciseData?.equipment || '',
          gifUrl: exerciseData?.gifUrl || '', // Animation URL
          instructions: exerciseData?.instructions || exercise.notes
        };
      } catch (error) {
        return exercise;
      }
```

```
    }));

    return {
      ...day,
      exercises: enrichedExercises
    };
  }));

  return enrichedDays;
}

async function fetchExerciseData(exerciseName: string) {
  try {
    const response = await
axios.get('https://exercisedb.p.rapidapi.com/exercises/name/' +
exerciseName.toLowerCase(), {
      headers: {
        'X-RapidAPI-Key': process.env.RAPIDAPI_KEY,
        'X-RapidAPI-Host': 'exercisedb.p.rapidapi.com'
      }
    });

    const exercise = response.data[0];
    return {
      targetMuscles: [exercise.target, exercise.bodyPart],
      equipment: exercise.equipment,
      gifUrl: exercise.gifUrl,
      instructions: exercise.instructions
    };
  } catch (error) {
    console.error('RapidAPI exercise fetch failed:', error);
    return null;
  }
}

function generateWorkoutPlanRuleBased(input: WorkoutGenerationInput) {
  // Fallback to pre-built templates
  const template = selectWorkoutTemplate(input.user.goals,
input.user.experience, input.user.availableDays);

  // Apply progressive overload if previous report exists
  if (input.previousReport) {
    applyProgressiveOverload(template, input.previousReport);
  }

  const endDate = new Date(input.startDate);
  endDate.setDate(endDate.getDate() + input.duration * 7);

  return new WorkoutPlan({
    userId: input.userId,
    name: template.name,
    duration: input.duration,
    startDate: input.startDate,
    endDate: endDate,
```

```
    status: 'active',
    days: template.days,
    generatedFrom: 'manual'
  }).save();
}
```

16.5. RapidAPI Integration

Available APIs for Fitness Platform:

1. **Nutritionix API** (Nutrition Data)
   - Endpoint: `https://nutritionix-api.p.rapidapi.com/v1_1/search`
   - Use: Get accurate nutrition facts for foods
2. **ExerciseDB** (Exercise Database)
   - Endpoint: `https://exercisedb.p.rapidapi.com/exercises`
   - Use: Get 1300+ exercises with animations, instructions, target muscles
3. **Edamam Nutrition Analysis** (Alternative)
   - Endpoint: `https://edamam-nutrition-analysis.p.rapidapi.com/api/nutrition-data`
   - Use: Analyze recipes and get detailed nutrition
4. **Spoonacular** (Recipe & Meal Planning)
   - Endpoint: `https://spoonacular-recipe-food-nutrition-v1.p.rapidapi.com/recipes/complexSearch`
   - Use: Find recipes, meal plans, grocery lists

Setup:

```
# Get free API key from: https://rapidapi.com
# Add to .env:
RAPIDAPI_KEY=your_rapidapi_key
```

16.6. Cron Job Integration

Update: `services/scheduler.ts`

```
import cron from 'node-cron';
import { generateDailyDietPlan } from './dietGenerationService';
import { User } from '../models/User';
import { ProgressLog } from '../models/ProgressLog';

// Run daily at 12:01 AM
cron.schedule('1 0 * * *', async () => {
```

```
    console.log('🚀 Starting daily diet generation...');

    const today = new Date();
    today.setHours(0, 0, 0, 0);

    const yesterday = new Date(today);
    yesterday.setDate(yesterday.getDate() - 1);

    // Get all active users
    const activeUsers = await User.find({
      'subscription.status': 'active',
      'subscription.expiresAt': { $gte: today }
    });

    console.log(`📊 Generating plans for ${activeUsers.length} users...`);

    for (const user of activeUsers) {
      try {
        // Get yesterday's progress
        const previousProgress = await ProgressLog.findOne({
          userId: user._id,
          date: yesterday
        });

        // Calculate adherence score
        const adherenceScore = previousProgress
          ? calculateAdherence(previousProgress)
          : 100; // Default for new users

        // Generate diet plan
        await generateDailyDietPlan({
          userId: user._id,
          date: today,
          user: {
            age: user.profile.age,
            weight: user.profile.weight,
            height: user.profile.height,
            gender: user.profile.gender,
            goals: user.profile.goals,
            activityLevel: user.profile.activityLevel,
            preferences: user.profile.preferences,
            restrictions: user.profile.restrictions
          },
          previousDayProgress: previousProgress ? {
            id: previousProgress._id,
            adherenceScore,
            totalCalories: calculateTotalCalories(previousProgress),
            missedMeals: findMissedMeals(previousProgress)
          } : undefined
        });

        console.log(`✅ Generated plan for user ${user._id}`);
      } catch (error) {
        console.error(`❌ Failed to generate plan for user ${user._id}:`,
```

```
error);
    }
  }

  console.log('□ Daily diet generation completed!');
});

function calculateAdherence(progressLog) {
  const plannedMeals = 4; // Typical daily meals
  const loggedMeals = progressLog.meals?.length || 0;
  return Math.min((loggedMeals / plannedMeals) * 100, 100);
}

function calculateTotalCalories(progressLog) {
  return progressLog.meals?.reduce((sum, meal) => sum + (meal.calories ||
0), 0) || 0;
}

function findMissedMeals(progressLog) {
  const expectedMeals = ['Breakfast', 'Lunch', 'Dinner', 'Snack'];
  const loggedMealNames = progressLog.meals?.map(m => m.mealName) || [];
  return expectedMeals.filter(meal => !loggedMealNames.includes(meal));
}
```

16.7. API Rate Limits & Cost Management

OpenRouter Free Tier:

- □ Free models available (Llama, Gemini Flash, Mistral)
- ⚠ Rate limits: ~60 requests/minute
- □ Strategy: Queue generation jobs, batch process users

RapidAPI Free Tier:

- □ Most APIs offer 100-500 free requests/month
- ⚠ Rate limits vary by API
- □ Strategy: Cache results, use sparingly for enrichment

Cost Optimization:

```
// Cache nutrition data to reduce API calls
import NodeCache from 'node-cache';
const nutritionCache = new NodeCache({ stdTTL: 86400 }); // 24 hour cache

async function fetchNutritionData(foodName: string) {
  const cacheKey = foodName.toLowerCase();
  const cached = nutritionCache.get(cacheKey);

  if (cached) {
    console.log('□ Cache hit for:', foodName);
    return cached;
```

```
  }

  const data = await fetchFromAPI(foodName);
  nutritionCache.set(cacheKey, data);
  return data;
}
```

16.8. Error Handling & Fallbacks

```
// Multi-tier fallback system
export async function generateDietPlanWithFallback(input) {
  try {
    // Tier 1: AI-powered generation (OpenRouter)
    return await generateDailyDietPlan(input);
  } catch (error) {
    console.warn('Tier 1 failed, trying Tier 2...', error);

    try {
      // Tier 2: Template-based with RapidAPI enrichment
      return await generateFromTemplatesWithEnrichment(input);
    } catch (error2) {
      console.warn('Tier 2 failed, using Tier 3...', error2);

      // Tier 3: Pure rule-based (always works)
      return await generateDietPlanRuleBased(input);
    }
  }
}
```

16.9. Testing the Generation System

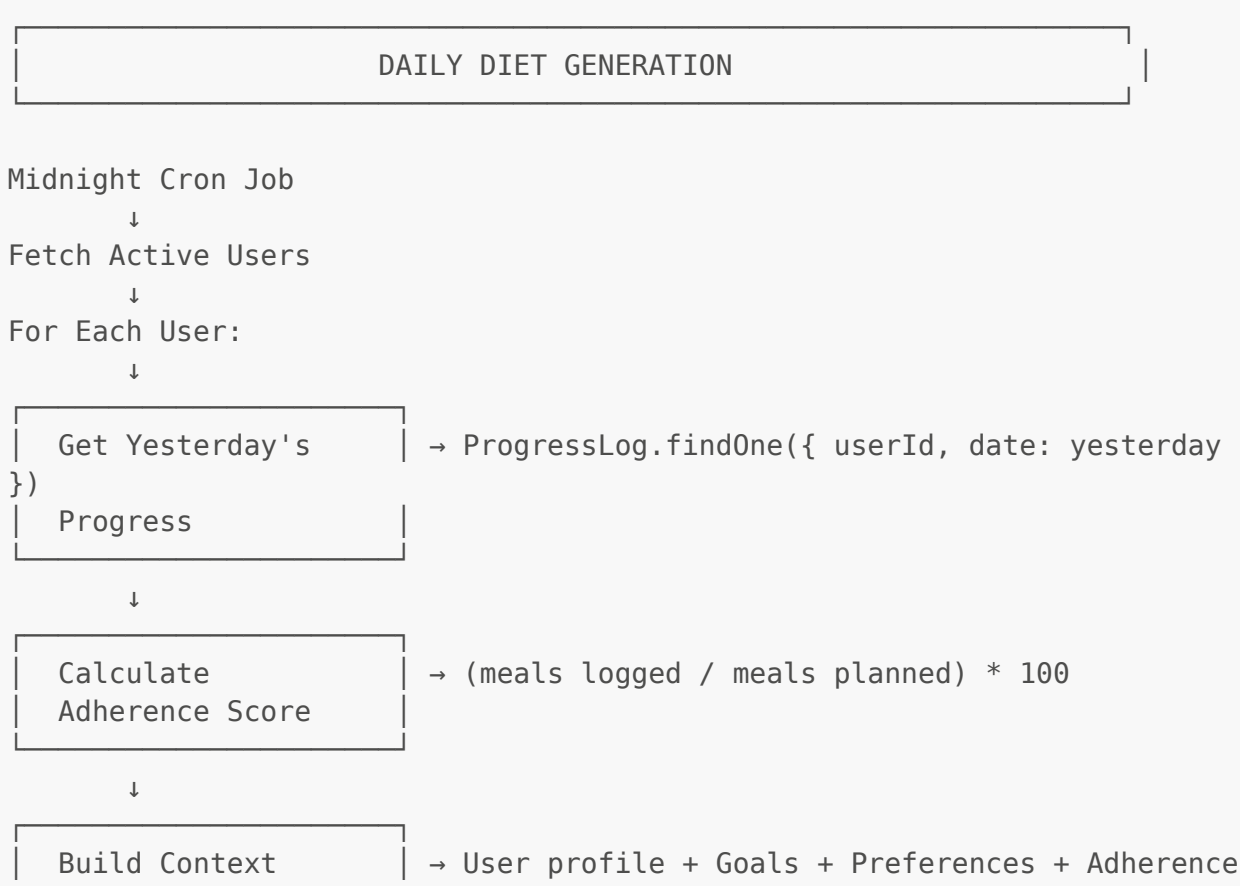
Test File: tests/generation.test.ts

```
import { generateDailyDietPlan } from '../services/dietGenerationService';
import { generateWorkoutCycle } from
'../services/workoutGenerationService';

describe('Diet Plan Generation', () => {
  it('should generate a diet plan for weight loss', async () => {
    const plan = await generateDailyDietPlan({
      userId: 'test123',
      date: new Date(),
      user: {
        age: 30,
        weight: 80,
        height: 175,
        gender: 'male',
        goals: ['weight-loss'],
        activityLevel: 'moderate'
```

```
      }
    });

    expect(plan.dailyCalories).toBeLessThan(2500);
    expect(plan.meals).toHaveLength(4);
    expect(plan.generatedFrom).toBe('ai');
  });

  it('should apply previous day adjustments', async () => {
    const plan = await generateDailyDietPlan({
      userId: 'test123',
      date: new Date(),
      user: { /* ... */ },
      previousDayProgress: {
        adherenceScore: 30, // Low adherence
        totalCalories: 1200,
        missedMeals: ['Dinner']
      }
    });

    // Should increase calories to make it easier
    expect(plan.dailyCalories).toBeGreaterThan(1800);
  });
});
```

---

19. Plan Generation Flow Diagram



```
│  for AI                │
└────────────────────────┘
        ↓
┌────────────────────────┐
│  OpenRouter API        │ → Generate meal plan using Gemini/Llama (FREE)
│  (Free Model)          │    Prompt includes adherence-based adjustments
└────────────────────────┘
        ↓
┌────────────────────────┐
│  RapidAPI              │ → Enrich meals with accurate nutrition data
│  (Nutritionix)         │    Cache results to save API calls
└────────────────────────┘
        ↓
┌────────────────────────┐
│  Save DietPlan         │ → DietPlan.create({ date, meals, generatedFrom:
'ai' })
│  to Database           │
└────────────────────────┘
        ↓
     Success □



┌──────────────────────────────────────────────────────────────────────────┐
│                    WORKOUT CYCLE GENERATION                              │
└──────────────────────────────────────────────────────────────────────────┘

User Requests New Cycle (or Previous Cycle Ends)
        ↓
POST /workouts/generate-cycle { duration: 8, userId }
        ↓
┌────────────────────────┐
│  Get Previous          │ → MonthlyWorkoutReport.findOne({ userId, year,
month })
│  Month Report          │
└────────────────────────┘
        ↓
┌────────────────────────┐
│  Analyze Strength      │ → strengthGains: [{ exercise, +10% weight }]
│  Progression           │
└────────────────────────┘
        ↓
┌────────────────────────┐
│  Build Workout         │ → User profile + Goals + Experience + Previous
gains
│  Context for AI        │    "Apply progressive overload: Bench Press 80kg
→ 88kg"
└────────────────────────┘
        ↓
┌────────────────────────┐
│  OpenRouter API        │ → Generate 8-week periodized program
│  (Llama 3.1)           │    With progressive overload built in
└────────────────────────┘
        ↓
```

```
┌─────────────────────────┐
│  RapidAPI               │ → Enrich exercises with animations, instructions
│  (ExerciseDB)           │    Get proper form cues and target muscles
└─────────────────────────┘
        ↓
┌─────────────────────────┐
│  Save WorkoutPlan       │ → WorkoutPlan.create({ duration, startDate,
endDate })
│  to Database            │
└─────────────────────────┘
        ↓
      Success □
```

Summary of Changes (Based on Latest Requirements)

This document has been updated to reflect the new **AI-powered intelligent plan generation system**:

New Features:

1. **AI-Powered Daily Diet Generation**:
   - **OpenRouter integration** with free models (Gemini Flash, Llama 3.1, Mistral)
   - Automated system generates personalized diet plans every day
   - Context-aware prompts include:
     - Previous day's adherence and progress
     - User's profile, goals, preferences, restrictions
     - Adaptive calorie/macro adjustments
   - **RapidAPI enrichment** for accurate nutrition data
   - Rule-based fallback if AI fails
2. **AI-Powered Workout Generation**:
   - Smart workout cycle generation using Llama 3.1
   - Duration-based programs (4-12 weeks) with periodization
   - **Progressive overload** based on previous cycle's strength gains
   - **ExerciseDB integration** for 1300+ exercises with GIF animations
   - Automatic cycle transitions with intelligent progression
3. **Monthly Reporting System**:
   - Aggregates all daily diet plans and workout progress into monthly summaries
   - Provides adherence scores, averages, and trends
   - Serves as historical data for future plan generation (feedback loop)
   - MongoDB aggregation pipelines for efficient computation
4. **Automated Scheduling**:
   - Cron jobs for daily diet generation (midnight, timezone-aware)

- Monthly report aggregation (1st of each month)
- Workout cycle completion checks and next cycle triggers
- Multi-tier fallback system for reliability

5. **Cost-Free AI Stack**:
   - □ OpenRouter free models (no usage limits for now)
   - □ RapidAPI free tier (100-500 requests/month per API)
   - □ Intelligent caching to minimize API calls
   - □ Rule-based fallbacks for zero-dependency operation

Schema Changes:

- **DietPlan**: Added `date`, `generatedFrom`, `previousDayProgressId`, `notes`
- **WorkoutPlan**: Added `duration`, `startDate`, `endDate`, `status`, `generatedFrom`
- **New Models**: `MonthlyDietReport`, `MonthlyWorkoutReport`

New API Endpoints:

- POST `/diet/generate-daily` - Automated daily generation (AI-powered)
- POST `/workouts/generate-cycle` - Duration-based cycle generation (AI-powered)
- GET `/reports/diet/monthly/:year/:month` - Monthly diet report
- GET `/reports/workout/monthly/:year/:month` - Monthly workout report
- POST `/reports/generate` - Manual report generation

Technology Stack:

- **AI**: OpenRouter (free models: Gemini Flash, Llama 3.1, Mistral 7B)
- **Data Enrichment**: RapidAPI (Nutritionix, ExerciseDB, Spoonacular)
- **Caching**: node-cache (in-memory) or Redis (production)
- **Scheduling**: node-cron
- **Fallback**: Rule-based template system

Implementation Priority:

1. □ Core CRUD (User, WorkoutPlan, DietPlan, ProgressLog)
2. □ Set up OpenRouter and RapidAPI integration
3. □ Monthly report schemas and aggregation logic
4. □ AI-powered diet generation service
5. □ AI-powered workout generation service
6. □ Cron job setup with timezone support
7. □ Caching layer and rate limiting
8. □ Frontend integration (hooks, pages, UI for reports)

Quick Start Guide:

1. Get free API keys:
   - OpenRouter: https://openrouter.ai/ (instant, no credit card)
   - RapidAPI: https://rapidapi.com/ (free tier: 100-500 req/month)
2. Install: `npm install openai axios node-cron node-cache`

3. Copy service files from Section 16 above
4. Set up cron jobs from Section 5
5. Test generation endpoints
6. Deploy!

If you want, I can now scaffold the backend skeleton (Express + Mongoose + auth routes) in a new folder `fitflow-api/` under the workspace and wire the `lib/api.ts` client in the frontend to point to it. Would you like me to scaffold now?