FitFlow Backend Implementation Roadmap (Express + MongoDB)

Date: 2025-10-31
Status: Draft for review
Owner: Backend/Full-stack Team

Goals

- Deliver a secure, scalable REST API for FitFlow features (users, workouts, diets, progress, analytics)
- Integrate cleanly with the existing Next.js frontend (App Router)
- Provide a smooth local dev experience (Docker) and straightforward deployment flow (Render/Fly/EC2)

High-level Architecture

- Frontend: Next.js (existing) on Vercel (or any Node host)
- Backend: Node.js + Express REST API
- DB: MongoDB (Atlas in prod, local Docker in dev)
- ODM: Mongoose
- Auth: JWT (access + refresh) with httpOnly cookies (preferred) or Authorization Bearer
- Validation: Zod (or Joi)
- Docs: Swagger/OpenAPI (via swagger-ui-express)
- Observability: Winston logs + request logging + health checks
- Security: Helmet, CORS, rate limiting, input sanitization, bcrypt
- Optional accelerators: Redis cache (analytics), BullMQ (jobs), S3-compatible storage (images)

```
+------------------+        HTTPS        +--------------------+
Private        +--------------+
|     Next.js      | <-----------------> |  Express API App   | <---------
----------> |   MongoDB    |
|  (Vercel/Node)   |                     |  (Node + Mongoose) |
|  (Atlas/Docker)
+------------------+                     +--------------------+
+--------------+
         ^                                            |
         |                                            | metrics/logs
         v                                            v
   Local dev uses Docker: API + MongoDB + (optional) Redis
```

Environments

- Local: Docker Compose (api + mongodb + optional redis)
- Staging: Same as prod on smaller tier (seeded demo data)
- Prod: MongoDB Atlas + managed Node host (Render/Fly/EC2) with HTTPS and environment secrets

---

Project Structure (backend)

```
fitflow-api/
├─ src/
│  ├─ app.ts                  # Express app wiring (middleware, routes)
│  ├─ server.ts               # HTTP server bootstrap
│  ├─ config/
│  │  ├─ env.ts               # env loading + validation
│  │  └─ db.ts                # mongoose connect
│  ├─ middleware/
│  │  ├─ auth.ts              # auth guard, role guard
│  │  ├─ error.ts             # error handler
│  │  ├─ validate.ts          # zod validator wrapper
│  │  └─ security.ts          # helmet, cors, rate-limit
│  ├─ models/
│  │  ├─ User.ts
│  │  ├─ WorkoutPlan.ts
│  │  ├─ DietPlan.ts
│  │  ├─ ProgressLog.ts
│  │  └─ Session.ts           # refresh token/session (optional)
│  ├─ routes/
│  │  ├─ auth.routes.ts
│  │  ├─ users.routes.ts
│  │  ├─ workouts.routes.ts
│  │  ├─ diet.routes.ts
│  │  ├─ progress.routes.ts
│  │  └─ analytics.routes.ts
│  ├─ controllers/
│  │  ├─ auth.controller.ts
│  │  ├─ users.controller.ts
│  │  ├─ workouts.controller.ts
│  │  ├─ diet.controller.ts
│  │  ├─ progress.controller.ts
│  │  └─ analytics.controller.ts
│  ├─ services/
│  │  ├─ auth.service.ts
│  │  ├─ users.service.ts
│  │  ├─ workouts.service.ts
│  │  ├─ diet.service.ts
│  │  ├─ progress.service.ts
│  │  └─ analytics.service.ts
│  ├─ schemas/                # zod schemas (req/resp)
│  ├─ utils/
│  │  ├─ jwt.ts
│  │  ├─ passwords.ts
│  │  ├─ logger.ts
│  │  └─ http.ts
│  └─ docs/
│     └─ openapi.ts          # swagger spec assembly
├─ tests/                    # jest + supertest
├─ package.json
├─ tsconfig.json
├─ .env.example
├─ docker-compose.yml        # local dev stack
└─ Dockerfile
```

Data Model (Mongoose)

User

```
{
  _id: ObjectId,
  email: string (unique, indexed),
  passwordHash: string,
  name: string,
  role: 'user' | 'admin',
  profile: {
    age?: number,
    weight?: number,
    height?: number,
    gender?: 'male'|'female'|'other',
    goals?: string[]
  },
  subscription?: {
    plan?: string,
    status?: 'active'|'inactive',
    expiresAt?: Date
  },
  createdAt: Date,
  updatedAt: Date
}
```

WorkoutPlan

```
{
  _id: ObjectId,
  userId: ObjectId (ref User, indexed),
  name: string,
  days: [{
    day:
'monday'|'tuesday'|'wednesday'|'thursday'|'friday'|'saturday'|'sunday',
    exercises: [{
      name: string,
      sets: number,
      reps: string,       // e.g., "8-10"
      rest: number,       // seconds
      notes?: string
    }]
  }],
  createdAt: Date,
  updatedAt: Date
}
```

DietPlan

```
{
  _id: ObjectId,
  userId: ObjectId (ref User, indexed),
  name: string,
  dailyCalories: number,
  macros: { protein: number, carbs: number, fats: number },
  meals: [{
    name: string,
    time: string,       // 08:00
    calories: number,
    foods: [{ name: string, portion: string, calories: number, macros: {
p: number, c: number, f: number } }]
  }],
  createdAt: Date,
  updatedAt: Date
}
```

ProgressLog

```
{
  _id: ObjectId,
  userId: ObjectId (ref User, indexed),
  date: Date (indexed),
  workout: {
    day?: string,
    completedExercises?: number,
    totalExercises?: number,
    durationSec?: number
  },
  meals: [{
    mealName: string,
    loggedAt: Date,
    calories?: number,
    macros?: { p: number, c: number, f: number }
  }],
  createdAt: Date
}
```

Indexes:

- User.email (unique), User.role
- WorkoutPlan.userId, DietPlan.userId
- ProgressLog.userId + date (compound)

API Design (REST)

Base URL: `http://localhost:4000/api`

Auth

- POST `/auth/register` -> create user
- POST `/auth/login` -> set access/refresh (httpOnly cookies) or return tokens
- POST `/auth/refresh` -> rotate access token
- POST `/auth/logout` -> clear cookies / revoke session
- GET `/auth/me` (auth) -> current user profile

Users (admin)

- GET `/users` (admin)
- GET `/users/:id` (admin/self for own)
- POST `/users` (admin)
- PUT `/users/:id` (admin/self limited)
- DELETE `/users/:id` (admin)

Workouts

- GET `/workouts` (auth) -> current user plans (or admin filter by userId)
- GET `/workouts/:id` (auth)
- POST `/workouts/generate` (admin) -> AI integration hook
- POST `/workouts` (admin) -> manual create
- PUT `/workouts/:id` (admin)

Diet

- GET `/diet` (auth)
- GET `/diet/:id` (auth)
- POST `/diet/generate` (admin)
- POST `/diet` (admin)
- PUT `/diet/:id` (admin)

Progress

- GET `/progress` (auth) -> user progress timeline
- POST `/progress/workout` (auth) -> log workout completion
- POST `/progress/meal` (auth) -> log meal
- GET `/progress/stats` (auth) -> aggregates

Analytics (admin)

- GET `/analytics/overview`
- GET `/analytics/user/:id`
- GET `/analytics/trends`

Pagination: `?page=1&limit=20`
Filtering: consistent query params (e.g., `?userId=...&from=...&to=...`)

Auth Strategy

Recommended: httpOnly cookie tokens (best UX with Next.js)

- Access token (short TTL, e.g., 15m)
- Refresh token (long TTL, e.g., 7d)
- Store refresh token id in DB (Session) for rotation/revocation
- CSRF protection: double-submit cookie or same-site=strict + only same-origin requests from Next.js

Alternative: Bearer tokens in Authorization header (simpler but store carefully on client)

Roles: `user`, `admin`

- Auth middleware sets `req.user`
- Role guard middleware ensures admin access to admin routes

Password: `bcrypt` with proper salt rounds

Validation & Errors

- Zod schemas per route to validate `req.body`, `req.query`, `req.params`
- Centralized error handler returning JSON envelope:

```
{ "ok": false, "error": { "code": "VALIDATION_ERROR", "message": "...",
"details": [...] } }
```

- Map domain errors to proper HTTP status codes

Security

- Helmet (sane defaults)
- CORS: allow `NEXT_PUBLIC_API_BASE_URL` origin(s)
- Rate limiting (e.g., 100 req/15m per IP)
- Input sanitization (xss-clean / express-validator sanitize or zod + escape)
- Strong password policy
- Disable x-powered-by, trust proxy set if behind load balancer

---

Observability

- Winston logger (JSON in prod), morgan for access logs in dev
- Health checks: `GET /health` (db ping + version)
- Request ID correlation (x-request-id)
- Basic metrics endpoint (future: Prometheus)

Local Dev & Tooling

Docker Compose (excerpt)

```
version: '3.9'
services:
  api:
    build: .
    ports:
      - "4000:4000"
    env_file:
      - .env
    depends_on:
      - mongo
  mongo:
    image: mongo:7
    restart: always
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db
volumes:
  mongo-data:
```

.env.example

```
PORT=4000
NODE_ENV=development
MONGODB_URI=mongodb://mongo:27017/fitflow
JWT_ACCESS_SECRET=changeme-access
JWT_REFRESH_SECRET=changeme-refresh
JWT_ACCESS_TTL=15m
JWT_REFRESH_TTL=7d
CORS_ORIGIN=http://localhost:3000
```

NPM scripts (backend)

```
- dev: ts-node-dev src/server.ts
- build: tsc
```

```
  - start: node dist/server.js
  - test: jest
```

Frontend Integration Plan

Frontend base URL: `process.env.NEXT_PUBLIC_API_BASE_URL` (e.g., http://localhost:4000/api)

API Client (`gym-app/lib/api.ts`)

- Add a small wrapper around fetch that:
  - Sends credentials when using cookie strategy: `credentials: 'include'`
  - Handles JSON parse, errors, and automatic refresh (optional)
  - Injects Authorization header if using Bearer tokens

Hooks to replace mock/localStorage

- `hooks/useAuth.ts` → /auth/login, /auth/me, /auth/logout
- `hooks/useWorkoutPlan.ts` → /workouts, /workouts/:id
- `hooks/useDietPlan.ts` → /diet, /diet/:id
- `hooks/useUserProgress.ts` → /progress, /progress/stats, POST logs

Recommended fetching strategy:

- SWR or React Query for caching/revalidation
- Use SSR selectively for SEO-less areas if needed; otherwise CSR with SWR is fine

Route Protection (Next.js)

- Next middleware (optional) to redirect unauthenticated users
- Or guard inside pages using `useAuth` state

CORS & Cookies

- If using cookies: set `credentials: 'include'` on client and `cors({ origin, credentials: true })` on server
- Set cookie flags: httpOnly, secure (prod), sameSite=strict

Testing Strategy

- Unit: services, utils (Jest)
- Integration: controllers/routes (supertest with in-memory Mongo or test DB)
- E2E (optional now): Playwright hitting local API and Next frontend
- Test data factories & seeds
- Coverage gate (min 80%)

Deployment Strategy

Option A (simple):

- Backend on Render/Fly/Heroku (Docker or Node buildpack)
- MongoDB Atlas
- Frontend on Vercel
- Configure env vars for each

Option B (DIY):

- Dockerized API on EC2 (or Lightsail) behind Nginx reverse proxy (HTTPS via Let’s Encrypt)
- MongoDB Atlas (managed) or self-hosted (not recommended for prod)

CI/CD (GitHub Actions):

- Lint + test on PR
- Build Docker image on main
- Deploy to environment via provider action (Render/Fly/EC2 SSH)

Performance & Scaling

- Use `.lean()` on read-heavy queries
- Proper indexes for filters and sort
- Pagination (limit/skip or keyset)
- Cache expensive analytics (Redis) with TTL
- Background jobs (BullMQ) for AI plan generation and aggregations
- Stress test with k6/Artillery before launch

Milestones & Timeline

Phase 0: Scaffolding (1-2 days)

- Initialize repo, Docker, Env validation, DB connect, health route

Phase 1: Auth & Users (3-4 days)

- Register, login, logout, refresh, me
- Role guard, hashing, validation
- Users CRUD (admin)

Phase 2: Plans (Workout & Diet) (4-6 days)

- CRUD + generation endpoints (stub AI)
- Hook to frontend generate pages

Phase 3: Progress & Analytics (4-6 days)

- Progress logs, stats, analytics overview
- Frontend hooks integrated

Phase 4: Hardening & Docs (2-3 days)

- Swagger, rate limit, logs, tests to 80% coverage
- Load test and fix hotspots

Acceptance Criteria

- All documented endpoints implemented with validation and RBAC
- Auth flow (login/logout/refresh/me) works end-to-end with httpOnly cookies
- Frontend pages use live API (no localStorage mocks)
- Swagger docs at `/api/docs`
- 80% test coverage on services/controllers
- Zero critical vulnerabilities, basic rate limiting enabled
- Monitoring: logs + health checks available

Open Questions

- AI plan generation provider (OpenAI vs local rules) and budget
- Payment/subscriptions scope now or later?
- Do we need multi-tenant or trainer orgs now?
- File uploads (profile pictures) now or defer?

Quick Start (Local Dev)

1. Clone both repos (frontend + backend).
2. Set `.env` from `.env.example`.
3. Run Docker Compose for DB.
4. Start API: `npm run dev`.
5. Start frontend: `npm run dev` with `NEXT_PUBLIC_API_BASE_URL=http://localhost:4000/api`.

This roadmap is designed to plug directly into the existing Next.js app with minimal churn. Once approved, we can scaffold the repo and begin Phase 0 immediately.