Fit Track Server

El servidor backend en Vapor para la app FitTrack.
Gestiona usuarios, entrenamientos y ejercicios.
Implementa autentificación con JWT.

Tecnologías principales:

- Vapor
- Fluent + SqLite
- JWT

Estructura de carpetas:

FitTrack_server/
|
|---Controllers/
|---DTOs/
|---Migrations/
|---Models/
|---Configure
|---Constants
|---routes
|---entrypoint
|---JWTToken

Modelos de dominio:

·User: campos principales(id, name, email, password, isAdmin).
Relación 1…many con resto de modelos.

·Exercise: campos principales(id, name, description, repetitions, sets, trainingID).
Relación con training many to one.

·Training: campos principales (id, name, start, end, userID)
Relación con exercise 1…many.

·Appointment: campos principales (id, date, trainer, userID)
Relación con con user many to one.

Notas clave sobre `Day`

1. `Day` funciona como **contenedor de Training**. Cada usuario puede tener varios días programados.

2. `Training` ahora tiene `dayID`, que indica **a qué día pertenece**.

3. Los **Exercise** y **ExerciseSet heredan implícitamente el día** del Training asociado.

4. Mantener la normalización: Day solo tiene ID y fecha; no guarda arrays de Trainings en la DB, pero en el modelo de Swift podés exponer `[Training]` como propiedad calculada.

5. Esto te permite fácilmente mostrar en UI: “Día Lunes 14 → lista de trainings → ejercicios → sets”.

DTOs:

En el proyecto diferenciamos entre **DTOs de entrada y DTOs de salida:**
·Los DTOs de entrada se utilizan para recibir datos del cliente, como UserLoginDTO en el proceso de login.

·Los DTOs de salida se utilizan para devolver datos al cliente de forma segura y estructurada, como UserDTO (info del usuario sin exponer contraseñas) o LoginResponseDTO (token tras la autenticación).

Así nos aseguramos de que nunca se exponen datos sensibles (passwordHash) y mantenemos claro qué datos se esperan en cada request/ response.

Autenticación:

Este servidor utiliza autenticación JWT (Json Web Tokens).

El archivo clave para esto es AuthController.swift, en la carpeta de Controllers.

Tokens generados:
-Access Token –> valido durante 24h (Constants.accessTokenLifeTime).
-Refresh Token –> válido durante 7 días (Constants.refreshTokenLifeTime).

Los campos del token, reflejados en Models/JWTToken.swift son:
· userID (UUID)
·userName
·expiration
·isRefresh (indica si el token es de acceso o refresh)

El flujo para este proceso quedaría del siguiente modo:

1. Registro → el usuario se crea con su contraseña hasheada (BCrypt).
2. Login → se verifica email y contraseña, se generan los tokens.
3. Refresh → con el refresh obtenemos un nuevo par de tokens.

Al usar JWT garantizamos que cada request se valide sin mantener la sesión en el servidor.

Middleware:

- AdminMiddleware: Encargado de asegurarse de que el usuario que realiza la solicitud es un coach.

- APIKeyMiddleware: Valida que la petición incluya una cabecera X-API-KEY válida. Pensado para restringir el acceso a ciertos endpoints a clientes que tengan la clave.

- RateLimitIPMiddleware: Limita el numero de peticiones que se pueden realizar en un minuto a 10. Tiene en cuenta la IP del usuario.
  Aplicado en las solicitudes que no necesitan de un usuario registrado, como auth/register/coach o auth/login.
- RateLimitUserMiddleware: Limita el número de peticiones que se pueden realizar en un minuto a 10. Tiene en cuenta el token del usuario.

Controllers:

Cada controlador implementa la lógica de un recurso/ modelo y sus endpoints REST.

·UserController

- GET /users → lista de usuarios.
- GET /users/:id → devuelve un usuario por ID.
- PUT /users/:id → Actualiza un usuario existente
- DELETE /users/:id → Elimina un usuario

·AuthController

- POST /auth/register → registro de usuarios
- POST /auth/login → login y generación de tokens.
- POST /auth/refresh → refresco de tokens.

·**TrainingController**

- POST /training -> registro de entrenamiento
- GET /trainings -> obtener todos los entrenamientos
- GET /trainingsByMonth ->obtener los entrenamientos programados para un mes específico.
- GET /training -> Obtener training por ID

    - PATCH /training -> Modificar un training
    - DELETE /training -> Eliminar un training

·ExerciseController

- Seguirán una estructura CRUD similar a UserController.
- Encargados de gestionar ejercicios, entrenamientos y citas respectivamente.

Todos los controladores se registrarán en routes.swift usando app.register(collection:).

Migraciones:

Las migraciones definen las tablas de la base de datos a partir de los modelos.

·CreateUserMigration:

- Crea columnas básicas para identificación: id, name, email, password.
- Añade campos opciones a profile: goal, age, weigth, heigth.
- Añade relación opcional ‘coach_id’ que referencia a otro usuario con el rol coach.
- Incluye timestamps ‘created_at’ y ‘updated_at’.
- La función revert elimina la tabla y sus campos.

·CreateTraining:

-

Rutas:

Todas las rutas de la API se configuran en routes.swift.
De este modo organizamos la API de forma modular y cada recurso tiene su grupo de rutas.

try app.register(collection: UserController())

try app.register(collection: AuthController())

try app.register(collection: ExerciseController())

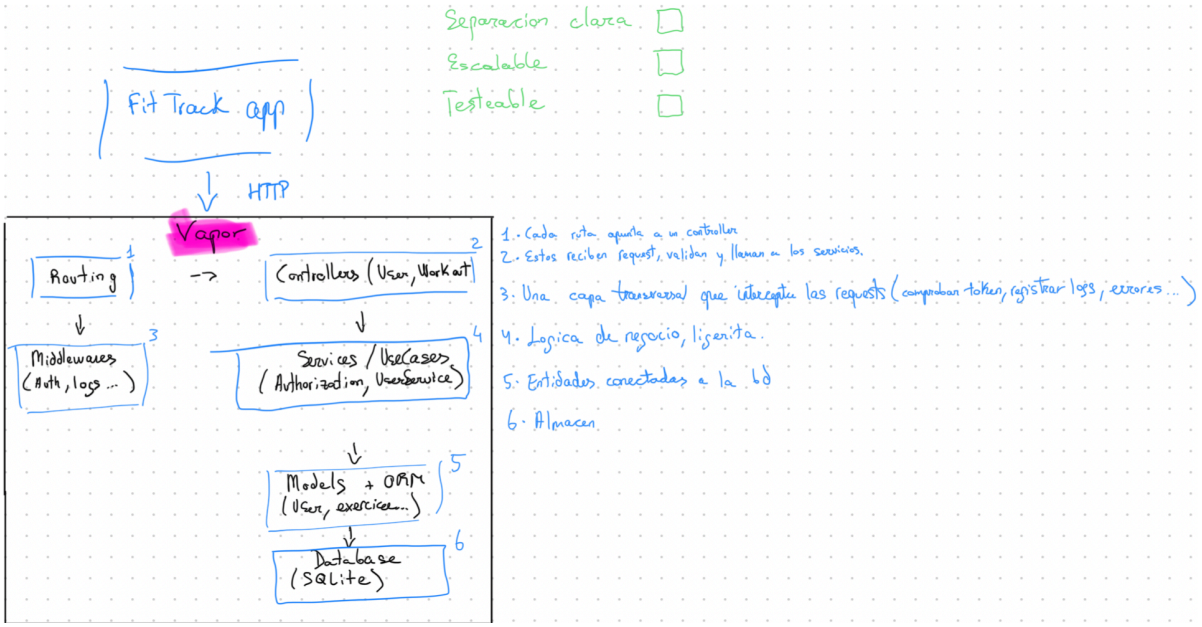
try app.register(collection: TrainingController())

Constants:

Centralización de parámetros de configuración, como el lifetime de los tokens.

TO-DO:

- Revisar los módelos ante las posibles implementaciones aprobadas desde diseño en las reuniones.
- Implementar algun middleware de autorizacion (como que el admin sea el único que pueda borrar usuarios o los Logs).
- Implementar el resto de controllers (exercise, workout, appointment).
- Testing unitario con DB en memoria. VaporTesting.



Realmente el mejor flujo de trabajo parece ir desarrollando del siguiente modo:
Modelos→ Controllers→ Rutas → Migraciones → Testing