Nama: Robani Diansyah

NPM: 23312246

Kelas: IF 23 Fx

1. Modul Utilitas Autentikasi

api/src/lib/auth.ts

```
import bcrypt from "bcryptjs";
import jwt, { SignOptions } from "jsonwebtoken";
import { NextRequest } from "next/server";
import { firebaseAdmin } from './firebase';

// Hash password dengan bcrypt
export async function hashPassword(password: string): Promise<string> {
    const saltRounds = 10;
    return await bcrypt.hash(password, saltRounds);
}

// Verifikasi password dengan hash
export async function verifyPassword(password: string, hashedPassword:
string): Promise<boolean> {
    return await bcrypt.compare(password, hashedPassword);
}

// Tipe data untuk isi token JWT
export interface JWTPayload {
    id: number;
    email: string;
    role: string;
    name: string;
}

// Generate JWT token
export function generateToken(payload: JWTPayload): string {
    const secret = process.env.JWT_SECRET;
    if (!secret) {
        throw new Error("JWT_SECRET tidak ada di file .env");
    }

    const expiresIn = process.env.JWT_EXPIRES_IN || "7d";

    return jwt.sign(payload, secret, {
        expiresIn: expiresIn as SignOptions['expiresIn'],
        issuer: "campushub-api",
    });
```

```
}

// Verifikasi JWT token
export function verifyToken(token: string): JWTPayload | null {
    try {
        const secret = process.env.JWT_SECRET;
        if (!secret) {
            throw new Error("JWT_SECRET tidak ada di file .env");
        }
        const decoded = jwt.verify(token, secret) as JWTPayload;
        return decoded;
    } catch {
        return null;
    }
}

// Ekstrak token dari Authorization header
export function extractToken(request: NextRequest): string | null {
    const authHeader = request.headers.get("authorization");
    if (!authHeader) return null;

    // Format: Bearer <token>
    const parts = authHeader.split(' ');
    if (parts.length !== 2 || parts[0] !== 'Bearer') {
        return null;
    }
    return parts[1];
}

// Dapatkan data user yang terautentikasi dari request
export function getAuthUser(request: NextRequest): JWTPayload | null {
    const token = extractToken(request);
    if (!token) return null;
    return verifyToken(token);
}

// Cek Role
export function hasRole(user: JWTPayload, allowedRoles: string[]): boolean {
    return allowedRoles.includes(user.role);
}

export const UNAUTHORIZED_RESPONSE = {
    success: false,
    error: 'Unauthorized',
    message: 'Token tidak valid atau sudah expired',
};

export const FORBIDDEN_RESPONSE = {
```

```
    success: false,
    error: 'Forbidden',
    message: 'Anda tidak memiliki akses ke halaman ini',
};

export function validateRequiredFields(
    body: Record<string, unknown>,
    fields: string[]
): string | null {
    const missingFields = fields.filter((field) => !body[field]);
    if (missingFields.length > 0) {
        return `Field ini wajib diisi: ${missingFields.join(', ')}`;
    }
    return null;
}

// Validasi format email
export function isValidEmail(email: string): boolean {
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return emailRegex.test(email);
}

// Validasi format password
export function validatePassword(password: string): string | null {
    if (password.length < 6) {
        return 'Password minimal 6 karakter';
    }
    return null;
}

// Tambahan: Verifikasi Token dari Frontend (Firebase)
export async function verifyFirebaseToken(token: string) {
    if (token === "TESTING_TOKEN") {
        return {
            uid: "user-dummy-123",
            email: "test@mahasiswa.com",
            name: "Mahasiswa Tester",
            picture: "https://via.placeholder.com/150"
        };
    }

    try {
        const decodedToken = await
firebaseAdmin.auth().verifyIdToken(token);
        return decodedToken;
    } catch (error) {
        console.error("Error verify firebase token:", error);
        return null;
```

```
    }
}
```

Deskripsi: File ini isinya fungsi-fungsi pendukung buat urusan keamanan di backend, biar kodingan di route utama nggak kepanjangan.

1. **verifyFirebaseToken**: Ini untuk ngecek validasi token yang dikirim sama Frontend. Jadi kita bisa pastiin kalau user yang login lewat Google itu beneran valid, bukan user palsu.
    - Catatan: Di situ saya tambahin sedikit logika buat TESTING_TOKEN. Tujuannya biar saya gampang ngetes API-nya manual lewat Postman/Apidog tanpa harus login lewat HP terus. Cuma buat testing aja.
2. **generateToken** & **verifyToken:** Ini fungsinya buat bikin token sesi (JWT). Jadi kalau user udah berhasil login, server bakal kasih kartu akses ini. Nanti kalau user mau akses data lain (kayak absen atau tugas), server tinggal cek token ini aja, nggak perlu bolak-balik ngecek database. Lebih hemat resource server.
3. **hashPassword**: Ini buat ngacak password (enkripsi) pakai bcrypt sebelum disimpan ke database. Ini sebenernya opsional karena kita fokus pakai Firebase, tapi saya siapin aja buat jaga-jaga kalau nanti ada fitur login biasa (tanpa Google), jadi datanya tetap aman.

2. Endpoint Registrasi & Sinkronisasi User

api/src/app/api/auth/register/route.ts

```
import { NextResponse } from "next/server";
import { verifyFirebaseToken, generateToken } from '@/lib/auth';
import Prisma from "@/lib/prisma";

export async function POST(request: Request) {
    try {
        const body = await request.json();
        // Frontend kirim "token" (firebase id token) & data pelengkap
        const { token, role, npm, phone, name } = body;

        if (!token) {
            return NextResponse.json(
                { success: false, message: "Token Firebase wajib dikirim" },
                { status: 400 }
            );
        }

        // Verifikasi Token ke Firebase
        const firebaseUser = await verifyFirebaseToken(token);
        if (!firebaseUser || !firebaseUser.email) {
            return NextResponse.json(
                { success: false, message: "Token Firebase tidak valid atau
expired" },
```

```
            { status: 401 }
        );
    }

    const email = firebaseUser.email;

    // 2. Cek apakah user ini udah ada
    let user = await Prisma.user.findUnique({
        where: { email },
    });

    // Kalau belum ada, kita BUAT BARU
    if (!user) {
        // Default role student kalau tidak dikirim
        const userRole = role || 'student';

        user = await Prisma.user.create({
            data: {
                email: email,
                name: name || firebaseUser.name || "No Name",
                password: "FIREBASE_AUTH_USER",
                role: userRole,
                npm: npm || null,
                phone: phone || null,
                firebaseUid: firebaseUser.uid,
            }
        });
    }

    // Generate Token JWT Backend (Session sendiri)
    const backendToken = generateToken({
        id: user.id,
        email: user.email,
        role: user.role,
        name: user.name
    });

    return NextResponse.json({
        success: true,
        message: "Login/Register berhasil",
        data: {
            user: {
                id: user.id,
                email: user.email,
                role: user.role,
                name: user.name
            },
            token: backendToken // Token ini yang disimpan frontend
```

```
            }
        });

    } catch (error) {
        console.error("Auth Error:", error);
        return NextResponse.json(
            { success: false, message: "Terjadi kesalahan server" },
            { status: 500 }
        );
    }
}
```

Deskripsi: Endpoint ini fungsinya buat pintu masuk (Login/Register). Bedanya sama login biasa, di sini saya nggak minta password ke user. Soalnya user kan loginnya pakai akun Google (lewat Firebase) di aplikasi mobile-nya. Jadi endpoint ini tugasnya cuma nyinkronin data aja.

Alur Kerja:

1. **Validasi Token dari Frontend** Pertama, backend nunggu kiriman token (ID Token) dari Frontend. Token ini bukti kalau user udah sukses login Google. Backend lalu ngecek ke server Firebase: "Ini token beneran valid atau nggak?".
2. **Cek Database Kita** Kalau token valid, sistem bakal ambil email user dari token itu, terus ngecek ke database PostgreSQL kita (pakai Prisma). Email ini udah terdaftar belum di sistem kampus.
3. **Otomatis Dibuatin Akun (Kalau User Baru)**
    - **Kalau belum ada:** Sistem langsung buatin akun baru secara otomatis. Data kayak Email dan UID (User ID) diambil dari Firebase, terus data pelengkap kayak NPM atau No HP diambil dari inputan user.
    - **Penyimpanan UID:** Saya juga simpan firebaseUid ke database. Biar nanti gampang kalau mau nyambungin data user di database kita sama data user di Firebase.
4. **Kasih Token Akses (JWT)** Terakhir, kalau user udah diverifikasi (mau dia user baru atau lama), backend bakal bikinin Token Sesi (JWT) punya kita sendiri. Token inilah yang nanti dipakai Frontend buat akses fitur-fitur lain kayak absen atau lihat tugas.

3. Struktur Database

prisma/schema.prisma

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema
```

```
// Looking for ways to speed up your queries, or scale easily with your
serverless or edge functions?
// Try Prisma Accelerate: https://pris.ly/cli/accelerate-init

generator client {
  provider = "prisma-client-js"
  output   = "../src/generated/prisma"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

// Model User untuk Dosen dan Mahasiswa
model User {
  id Int @id @default(autoincrement())  // ID unik buat tiap user
  name String                           //Nama lengkap user
  email String @unique                  // Email user, @unique artinya gak
boleh ada yang sama
  password String?                      //Password diHash (diamanin)
  role String @default("student")       // Peran user, kalau gak diisi
otomatis jadi "student
  npm String?                           // Tanda tanya (?) artinya gak wajib
diisi (opsional), buat mahasiswa
  phone String?                         // Nomor HP, juga gak wajib diisi
  avatar String?                        // Link ke foto profil, ini juga gk
wajib diisi
  createdAt DateTime @default(now())    // Kapan akun dibuat
  updatedAt DateTime @updatedAt          // Kapan data user terakhir diubah

  firebaseUid String? @unique

  // Relasi ke tabel lain
  tasks Task[]                      // Untuk banyak tugas
  attendances Attendance[]          // ..... banyak data absensi
  products Product[]                // ..... ..... produk
  transactions Transaction[]        // ..... ..... transaksi
  notifications Notification[]      // ..... ..... notifikasi
  qrSessionsCreated QRSession[]    @relation("LecturerSessions")

  // Indeks untuk mempercepat pencarian data di database
  @@index([email])  // Cari user berdasarkan email
  @@index([role])   // Cari user berdasarkan role
  @@index([firebaseUid])
}

// Model Task untuk mengatur tugas-tugas kuliah
```

```prisma
model Task {
  id Int @id @default(autoincrement())
  title String                              // Untuk judul tugas
  description String? @db.Text              // Deskripsi tugas
  course String?                            // Tugas ini buat mata kuliah apa
  dueDate DateTime                          // Deadline atau tanggal
pengumpulan
  priority String @default("medium")        // Tingkat prioritas (low, medium,
high)
  status String @default("todo")            // Status pengerjaan
  reminder DateTime?                        // Untuk notif
  createdAt DateTime @default(now())        // Kapan tugas dibuat
  updatedAt DateTime @updatedAt              // Kapan tugas terakhir diubah

  // Relasi ke tabel user
  user User @relation(fields: [userId], references: [id], onDelete: Cascade)
// Tugas ini punya siapa
  userId Int // Kunci buat nyambungin ke ID di tabel user

  // Indexing buat percepat query
  @@index([userId])  // Biar cepet nyari semua tugas milik satu user
  @@index([status])  // Biar cepet filter tugas berdasarkan status
  @@index([dueDate]) // Biar cepet ngurutin tugas berdasarkan deadline
}

// Model Attendance untuk simpan absen mahasiswa
model Attendance {
  id Int @id @default(autoincrement())
  courseName String                       // Absen matakuliah apa
  qrHash String                           // QR yang di scan mahasiswa
  latitude Float?                         // Koordinat mahasiswa pas absen
  longitude Float?                         // Sama aja
  status String @default("present")       // Status absennya apakah(masuk,
alpa, telat)
  timestamp DateTime @default(now())      // Jam berapa dia absen

  // Relasi ke tabel user
  user   User @relation(fields: [userId], references: [id], onDelete:
Cascade) // Siapa yang absen?
  userId Int  // Kunci tamu ke tabel User

  // Indexing
  @@index([userId])   // Biar cepet nyari riwayat absen satu mahasiswa
  @@index([qrHash])   // Buat validasi QR
  @@index([timestamp])// Biar cepet ngurutin absen berdasarkan waktu
}

// Model QRSession yang di buat oleh dosen
```

```
model QRSession {
  id Int @id @default(autoincrement())
  courseName String                       // Sesi QR ini untuk matkul apa
  qrHash String @unique                   // Kode unik sesi ini nanti
ditampilin jadi QR gak boleh ada yang sama
  expiresAt DateTime                      // Kapan QR ini kadaluarsa / gk
bisa di scan lagi
  latitude Float?                         // Lokasi dosen / kelas pas bikin
absen
  longitude Float?                        // Sama aja
  radius Int @default(50)                 // Radius toleransi dari lokasi
dosen
  createdBy String                        // Email dosen yang buat QR nya
  createdAt DateTime @default(now())      // Sesi QR dibuat

  // Relasi ke User (Dosen) yang membuat sesi ini
  lecturer   User @relation("LecturerSessions", fields: [lecturerId],
references: [id], onDelete: Cascade)
  lecturerId Int

  // Indexing
  @@index([qrHash])    // Biar pas mahasiswa scan, nyari datanya cepet
  @@index([expiresAt]) // Biar gampang nyari sesi QR yang udah kadaluarsa
}

// model Product barang yang dijual di marketplace
model Product {
  id Int @id @default(autoincrement())
  name String                             // Nama produk
  description String? @db.Text            // Deskripsi produk
  price Int                               // Harga dari produk
  stock Int @default(0)                    // Stok yang tersedia
  category String @default("others")      // Kategori produk
  images String[]                          // Array karena untuk menyimpan
link dari gambar produk
  condition String @default("new")        // Kondisi dari barangnya (baru,
bagus, dll)
  status String @default("available")    // Status produknya (tersedia,
terjual, dll)
  createdAt DateTime @default(now())      // Kapan produk di upload
  updatedAt DateTime @updatedAt           // Kapan produknya di update

  // Relasi ke tabel user
  owner   User @relation(fields: [ownerId], references: [id], onDelete:
Cascade) // Siapa penjualnya?
  ownerId Int  // Kunci tamu ke tabel User

  transactions Transaction[] // Satu produk bisa ada di banyak transaksi
```

```
  // Indexing
  @@index([ownerId])  // Biar cepet nampilin semua produk dari satu penjual
  @@index([status])
  @@index([category])
}

// model Transaction untuk riwayat jual - beli
model Transaction {
  id Int @id @default(autoincrement())
  quantity Int @default(1)              // Banyak barang yang dibeli
  totalPrice Int                        // Total harga
  status String @default("pending")     // Status transaksi (pending,
sukses, batal)
  notes String? @db.Text                // Catatan dari sipembeli
  createdAt DateTime @default(now())    // Kapan transaksi dibuat
  updatedAt DateTime @updatedAt          // Kapan status transaksi di update

  // Relasi ke tabel user
  product   Product @relation(fields: [productId], references: [id],
onDelete: Cascade) // Produk apa yang dibeli?
  productId Int     // Kunci tamu ke tabel Product

  buyer   User @relation(fields: [buyerId], references: [id], onDelete:
Cascade) // Siapa yang beli?
  buyerId Int  // Kunci tamu ke tabel User

  // Indexing
  @@index([buyerId])
  @@index([productId])
  @@index([status])
}

// model Notification untuk notif di aplikasi
model Notification {
  id Int @id @default(autoincrement())
  title String                          // Judul notif
  body String @db.Text                  // Isi pesan notif nya
  type String @default("general")       // Tipe notif
  isRead Boolean @default(false)        // Cek udah di baca atau belum
  data Json?                            // Untuk simpan data tambahan
  createdAt DateTime @default(now())    // Kapan notif di buat

  // Relasi ke tabel user
  user   User @relation(fields: [userId], references: [id], onDelete:
Cascade) // Notif ini buat siapa?
  userId Int  // Kunci tamu ke tabel User
```

```
  // Indexing
  @@index([userId])   // Biar cepet nampilin semua notif buat satu user
  @@index([isRead])   // Biar cepet filter notif yang belum dibaca
  @@index([createdAt])
}
```

Deskripsi: File ini fondasi database aplikasi kita. Di sini saya atur tabel-tabel apa aja yang bakal dibikin di PostgreSQL biar datanya terstruktur dan nggak berantakan.

Poin-Poin Penting:

1. **Tabel User** sebagai Pusat Data Tabel User ini intinya. Di sini saya simpan data Dosen dan Mahasiswa. Yang paling krusial di sini ada kolom firebaseUid. Kolom ini fungsinya sebagai "jembatan" buat nyambungin user yang login pakai Google (Firebase) dengan data profil mereka di database kita.
2. **Fitur Akademik (Tugas & Absen)** Buat fitur utama kampus, saya siapin tabel Task (buat manajemen tugas) dan pasangan tabel QRSession & Attendance.
    - Alurnya: Dosen bikin sesi di QRSession, terus Mahasiswa scan QR-nya, dan datanya masuk ke Attendance.
3. **Ekspansi Fitur Marketplace** Biar aplikasinya lebih powerful, saya juga siapin tabel Product dan Transaction. Jadi sistemnya udah siap kalau nanti mau dipakai buat mahasiswa jual-beli barang bekas (kayak buku atau alat tulis) di dalam aplikasi.
4. **Optimasi Performa (Indexing)** Saya juga pasang @index di kolom-kolom yang sering dicari (kayak email, status tugas, atau qrHash). Tujuannya biar nanti kalau datanya udah ribuan, aplikasi tetep ngebut pas nyari data dan nggak lemot.

4. Endpoint Login User

api/src/app/api/auth/login/route.ts

```
import { NextResponse } from "next/server";
import { verifyFirebaseToken, generateToken } from "@/lib/auth";
import Prisma from "@/lib/prisma";

export async function POST(request: Request) {
    try {
        const body = await request.json();
        const { token } = body;

        if (!token) {
            return NextResponse.json(
                { success: false, message: "Token Firebase wajib dikirim" },
                { status: 400 }
            );
        }
```

```
        // Verifikasi Token ke Firebase
        const firebaseUser = await verifyFirebaseToken(token);
        if (!firebaseUser || !firebaseUser.email) {
            return NextResponse.json(
                { success: false, message: "Token tidak valid atau expired"
},
                { status: 401 }
            );
        }

        const email = firebaseUser.email;

        // Cari user di database berdasarkan email dari token
        const user = await Prisma.user.findUnique({
            where: { email },
            select: {
                id: true,
                name: true,
                email: true,
                role: true,
                npm: true,
                phone: true,
                avatar: true
            }
        });

        // Jika user tidak ketemu harus (register)
        if (!user) {
            return NextResponse.json(
                { success: false, message: "User belum terdaftar. Silakan
registrasi terlebih dahulu." },
                { status: 404 }
            );
        }

        // Kalau user ada Generate Token Session
        const sessionToken = generateToken({
            id: user.id,
            email: user.email,
            role: user.role,
            name: user.name
        });

        // Kembalikan respons sukses
        return NextResponse.json(
            {
                success: true,
```

```
                message: "Login berhasil",
                data: {
                    token: sessionToken,
                    user: user
                }
            },
            { status: 200 }
        );

    } catch (error) {
        console.error("Login Error:", error);
        return NextResponse.json(
            { success: false, message: "Terjadi kesalahan server saat login"
},
            { status: 500 }
        );
    }
}
```

Deskripsi: Kalau file Register tadi buat user baru, nah file ini khusus buat User Lama yang mau masuk aplikasi. Logic-nya lebih simpel karena kita nggak perlu simpan data baru, cuma ngecek data lama aja.

Alur Kerja:

1. **Terima & Verifikasi Token** Sama kayak register, langkah pertama backend nerima kiriman token Firebase dari Frontend. Kita cek dulu tokennya valid nggak ke server Google. Kalau valid, kita ambil alamat email yang ada di dalam token itu.
2. **Cari User di Database** dari email tadi, sistem langsung nyari di database PostgreSQL: "Ada nggak sih user yang emailnya ini?"
3. **Pengecekan Status (User Found vs Not Found)**
    - **Kalau User Nggak Ketemu:** Backend bakal langsung nolak dengan status **404**. Kita kasih pesan "User belum terdaftar", jadi dia harus register dulu. Ini buat nyegah user ilegal masuk.
    - **Kalau User Ketemu:** Berarti dia emang mahasiswa/dosen kita yang sah.
4. **Tukar Token (Token Exchange)** Karena user valid, backend bakal bikinin Token Sesi (JWT) baru buat dia. Token inilah yang dikirim balik ke Frontend buat disimpan. Jadi nanti kalau dia mau absen atau liat tugas, dia pakai token dari kita ini, bukan token dari Google lagi.

5. Helper Koneksi Database

api/src/lib/prisma.ts

```
import { PrismaClient, Prisma } from "@/generated/prisma";
```

```ts
// Buat "slot" global buat nyimpen koneksi Prisma
declare global {
    var prisma: PrismaClient | undefined;
}

// Buat koneksi Prisma
const prismaClientSingleton = () => {
    return new PrismaClient({
        log: process.env.NODE_ENV === 'development'
            ? ['query', 'error', 'warn']
            : ['error'],
    });
};

// Cek udah ada koneksi di global?
const prisma = globalThis.prisma ?? prismaClientSingleton();

export default prisma;

if (process.env.NODE_ENV !== 'production') {
    globalThis.prisma = prisma;
}

// Fungsi bantuan buat putusin koneksi (saat testing)
export async function disconnectPrisma() {
    await prisma.$disconnect();
}

// Helper function untuk handle database errors
export function handlePrismaError(error: unknown) {
    // Kita cek apakah error berasal dari Prisma
    if (error instanceof Prisma.PrismaClientKnownRequestError) {
        if (error.code === 'P2002') {
            const field = (error.meta?.target as string[])?.[0] || 'field';
            return {
                status: 400,
                message: `${field} sudah terdaftar di sistem`,
            };
        }
        if (error.code === 'P2025') {
            return {
                status: 404,
                message: 'Data tidak ditemukan'
            }
        }
        if (error.code === 'P2003') {
            return {
                status: 400,
```

```
                message: 'Data referensi tidak valid atau sudah terhapus'
            }
        }
    }

    console.error("Database Error:", error); // Log error
    return {
        status: 500,
        message: 'Terjadi kesalahan pada server database',
    };
}
```

Deskripsi: File ini fungsinya sebagai pusat kendali koneksi antara aplikasi Backend kita dengan database PostgreSQL. Daripada setiap file bikin koneksi sendiri-sendiri, mending semuanya lewat satu pintu ini biar susuai standar.

Alur Kerja:

1. **Penerapan Singleton Pattern (**Anti-Lemot**)** Di sini saya pakai teknik Singleton. Masalah utama di Next.js itu kalau kita lagi coding (Development mode), server sering restart sendiri (hot reload). Kalau nggak pakai cara ini, setiap restart server bakal bikin koneksi baru ke database. Lama-lama koneksinya numpuk (connection pool exhausted) dan bikin laptop/server jadi lemot banget.
    - Solusinya: Saya simpan koneksinya di variable globalThis. Jadi kalau server restart, dia bakal cek: "Udah ada koneksi belum?". Kalau udah ada, pakai yang lama. Kalau belum, baru bikin baru.
2. **Centralized Error Handling (handlePrismaError)** Seringkali error dari database itu bahasanya aneh dan susah dimengerti user (contoh: Error P2002). Makanya saya buat fungsi handlePrismaError di sini.
    - Fungsinya buat "menerjemahkan" kode error mesin menjadi bahasa manusia yang enak dibaca Frontend.
    - Contoh: Kalau muncul error **P2002**, sistem otomatis bilang "Data sudah terdaftar". Kalau **P2025**, dibilang "Data tidak ditemukan".
    - Ini bikin kodingan di file lain (Controller/Route) jadi jauh lebih bersih karena nggak perlu ngulang-ngulang try-catch yang panjang.

6. Konfigurasi Firebase Admin

api/src/lib/firebase.ts

```
import admin from 'firebase-admin';

if (!admin.apps.length) {
```

```
    if (process.env.FIREBASE_PROJECT_ID && process.env.FIREBASE_PRIVATE_KEY)
{
        try {
            admin.initializeApp({
                credential: admin.credential.cert({
                    projectId: process.env.FIREBASE_PROJECT_ID,
                    clientEmail: process.env.FIREBASE_CLIENT_EMAIL,
                    privateKey:
process.env.FIREBASE_PRIVATE_KEY?.replace(/\\n/g, '\n'),
                }),
            });
            console.log("Firebase Admin berhasil connect!");
        } catch (error) {
            console.error("Gagal init Firebase", error);
        }
    } else {
        console.warn("Firebase Admin TIDAK berjalan karena .env belum
lengkap. (Mode Testing Only)");
    }
}

export const firebaseAdmin = admin;
```

Deskripsi: File ini adalah jembatan penghubung antara Backend kita dengan server Google (Firebase). Tanpa file ini, fitur login via Google nggak bakal bisa diverifikasi sama backend.

Alur Kerja:

1. **Cegah Inisialisasi Ganda (Double Init Prevention)** Di Next.js, kalau kita lagi mode development (ngoding), server sering restart otomatis. Kalau kodingannya biasa aja, pasti bakal error "Firebase App already exists". Makanya saya pakai !admin.apps.length buat ngecek: "Eh, Firebase udah nyala belum?". Kalau udah, ya pakai yang ada, jangan nyalain ulang.
2. **Handling Private Key (.replace)** Ini trik khusus biasanya kalau simpan Private Key di file .env, format baris baru (\n) sering rusak atau kebaca sebagai teks biasa. Makanya saya tambah fungsi .replace(/\\n/g, '\n') biar kuncinya tetap valid dan bisa dibaca sama sistem.
3. **Mode "Fail-Safe" (Anti Crash)** Saya nambahin logika conditional (IF-ELSE). Kalau file .env-nya belum diisi (misal teman saya belum ngirim kuncinya), server nggak bakal crash/mati. Dia cuma bakal ngasih peringatan warning di terminal. Jadi backend tetap bisa jalan buat testing fitur lain yang nggak butuh Firebase.

7. Manajemen Tugas (Tasks)

api/src/app/api/tasks/route.ts

```ts
import { Prisma } from "@/generated/prisma";
import prisma from '@/lib/prisma'
import { getAuthUser, UNAUTHORIZED_RESPONSE, validateRequiredFields } from
"@/lib/auth";
import { NextRequest, NextResponse } from "next/server";

// Ambil semua tugas
export async function GET(request: NextRequest) {
    try {
        const authUser = getAuthUser(request)
        if (!authUser) {
            return NextResponse.json(
                { ...UNAUTHORIZED_RESPONSE },
                { status: 401 }
            )
        }

        // Ambil parameter query
        const { searchParams } = new URL(request.url)
        const status = searchParams.get('status')
        const priority = searchParams.get('priority')
        const course = searchParams.get('course')

        const where : Prisma.TaskWhereInput = {
            userId: authUser.id
        }
        if (status) where.status = status
        if (priority) where.priority = priority
        if (course) where.course = course

        // Ambil tasks
        const tasks = await prisma.task.findMany({
            where,
            orderBy: [
                { status: 'asc' },
                { dueDate: 'asc' },
                { createdAt: 'desc' }
            ],
            include: {
                user: {
                    select: {
                        id: true,
                        name: true,
                        email: true
                    }
                }
            }
        })
```

```
        // Hitung statistik
        const stats = {
            total: tasks.length,
            todo: tasks.filter(t => t.status === 'todo').length,
            inProgress: tasks.filter(t => t.status === 
'in_progress').length,
            done: tasks.filter(t => t.status === 'done').length
        }
        return NextResponse.json(
            { success: true, data: tasks, stats },
            { status: 200 }
        )
    } catch (error) {
        console.error("Get tasks error:", error)
        if (error instanceof Error) {
            console.error("Error message:", error.message)
        }
        return NextResponse.json(
            { success: false, error: "Internal Server Error", message: 
"Terjadi kesalahan saat mengambil data tasks" },
            { status: 500 }
        )
    }
}

// Buat tugas baru
export async function POST(request: NextRequest) {
    try {
        const authUser = getAuthUser(request)
        if (!authUser) {
            return NextResponse.json(
                { ...UNAUTHORIZED_RESPONSE },
                { status: 401 }
            )
        }

        // parsing request body
        const body = await request.json()
        const { title, description, course, dueDate, priority, reminder } = 
body

        // validasi fields (isi wajib)
        const validationError = validateRequiredFields(body, ['title'])
        if (validationError) {
            return NextResponse.json(
                { success: false, error: "Validation Error", message: 
validationError },
```

```
                { status: 400 }
            )
        }

        // Validasi priority
        const validPriorities = ['low', 'medium', 'high']
        const taskPriority = priority || 'medium'
        if (!validPriorities.includes(taskPriority)) {
            return NextResponse.json(
                { success: false, error: 'Invalid Priority', message:
'Priority harus low, medium, atau high' },
                { status: 400 }
            )
        }

        // Buat tugas
        const newTask = await prisma.task.create({
            data: {
                title,
                description: description || null,
                course: course || null,
                dueDate: new Date(dueDate),
                priority: taskPriority,
                reminder: reminder ? new Date(reminder) : null,
                userId: authUser.id
            },
            include: {
                user: {
                    select: {
                        id: true,
                        name: true,
                        email: true
                    }
                }
            }
        })
        return NextResponse.json(
            { success: true, message: "Tugas berhasil dibuat", data: newTask
},
            { status: 201 }
        )
    } catch (error) {
        console.error("Create task error:", error)
        if (error instanceof Error) {
            console.error("Error message:", error.message)
        }
        return NextResponse.json(
```

```
            { success: false, error: "Internal Server Error", message:
"Terjadi kesalahan saat membuat tugas" },
            { status: 500 }
        )
    }
}
```

Deskripsi: File ini berfungsi sebagai Terminal Utama untuk data tugas. Di sini saya menangani permintaan yang mengambil banyak data sekaligus atau pembuatan data baru.

Alur Kerja:

1. **Sistem Filter & Sorting (GET):** Agar data yang dikirim ke Frontend rapi, saya menggunakan searchParams. Jadi, user bisa memfilter tugas berdasarkan status (misal: "tampilkan yang belum selesai saja") atau prioritas. Selain itu, data otomatis diurutkan (orderBy) dengan prioritas:
    - Status (biar yang belum beres muncul paling atas).
    - Deadline (biar yang mepet tenggat waktu muncul duluan).
2. **Statistik Otomatis:** Di bagian respons GET, saya menyisipkan objek stats. Backend sekalian menghitung berapa jumlah tugas Todo, In Progress, dan Done. Tujuannya agar Frontend tidak perlu melakukan perhitungan manual lagi di sisi klien (HP), sehingga aplikasi terasa lebih ringan.
3. **Validasi Input (POST):** Saat membuat tugas baru, sistem memvalidasi inputan wajib seperti title. Saya juga membatasi input priority hanya boleh berisi low, medium, atau high untuk menjaga konsistensi data di database.

8. Manipulasi Detail Tugas

api/src/app/api/tasks/[id]/route.ts

```
import { NextResponse, NextRequest } from "next/server";
import prisma from "@/lib/prisma";
import { getAuthUser, UNAUTHORIZED_RESPONSE } from "@/lib/auth";

// Helper type untuk parameter ID
interface Params {
    params: { id: string }
}

// Ambil detail satu tugas
export async function GET(request: NextRequest, { params }: Params) {
    try {
        const user = getAuthUser(request);
        if (!user) return NextResponse.json(UNAUTHORIZED_RESPONSE, { status:
401 });
```

```
        const taskId = parseInt(params.id);

        const task = await prisma.task.findUnique({
            where: { id: taskId },
            include: {
                user: { select: { id: true, name: true, email: true } }
            }
        });

        if (!task || task.userId !== user.id) {
            return NextResponse.json({ success: false, message: "Tugas tidak
ditemukan" }, { status: 404 });
        }

        return NextResponse.json({ success: true, data: task });
    } catch (error) {
        console.error("GET Tasks Error:", error);
        return NextResponse.json({ success: false, message: "Server error"
}, { status: 500 });
    }
}

// Update tugas
export async function PATCH(request: NextRequest, { params }: Params) {
    try {
        const user = getAuthUser(request);
        if (!user) return NextResponse.json(UNAUTHORIZED_RESPONSE, { status:
401 });

        const taskId = parseInt(params.id);
        const body = await request.json();

        const existingTask = await prisma.task.findUnique({ where: { id:
taskId } });

        if (!existingTask || existingTask.userId !== user.id) {
            return NextResponse.json({ success: false, message: "Tugas tidak
ditemukan atau akses ditolak" }, { status: 404 });
        }

        const updatedTask = await prisma.task.update({
            where: { id: taskId },
            data: {
                title: body.title,
                description: body.description,
                status: body.status,
                priority: body.priority,
```

```
                dueDate: body.dueDate ? new Date(body.dueDate) : undefined,
                course: body.course,
                reminder: body.reminder ? new Date(body.reminder) : null
            }
        });

        return NextResponse.json({ success: true, message: "Tugas berhasil
diupdate", data: updatedTask });
    } catch (error) {
        console.error("PATCH Task Error:", error);
        return NextResponse.json({ success: false, message: "Gagal update
tugas" }, { status: 500 });
    }
}

// Hapus tugas
export async function DELETE(request: NextRequest, { params }: Params) {
    try {
        const user = getAuthUser(request);
        if (!user) return NextResponse.json(UNAUTHORIZED_RESPONSE, { status:
401 });

        const taskId = parseInt(params.id);

        const existingTask = await prisma.task.findUnique({ where: { id:
taskId } });

        if (!existingTask || existingTask.userId !== user.id) {
            return NextResponse.json({ success: false, message: "Tugas tidak
ditemukan" }, { status: 404 });
        }

        await prisma.task.delete({ where: { id: taskId } });

        return NextResponse.json({ success: true, message: "Tugas berhasil
dihapus" });
    } catch (error) {
        console.error("DELETE Task Error:", error);
        return NextResponse.json({ success: false, message: "Gagal menghapus
tugas" }, { status: 500 });
    }
}
```

Deskripsi: Berbeda dengan file sebelumnya, file ini berada di dalam folder dinamis [id]. Fungsinya spesifik untuk menangani satu tugas berdasarkan ID-nya. Ini adalah backend untuk fitur "Lihat Detail", "Edit", dan "Hapus".

Alur Kerja:

1. **Validasi Kepemilikan (Security Check):** Ini adalah fitur keamanan paling krusial. Di setiap fungsi (GET, PATCH, DELETE), saya selalu mengecek: if (task.userId !== user.id) Artinya, sistem memastikan bahwa yang sedang mencoba mengedit atau menghapus tugas adalah pemilik asli tugas tersebut. Jika Mahasiswa A mencoba mengutak-atik tugas milik Mahasiswa B (meskipun tahu ID-nya), sistem akan menolak aksesnya (Error 404/403).
2. **Update Fleksibel (PATCH):** Saya menggunakan method PATCH untuk pengeditan. Ini memungkinkan mahasiswa mengupdate data. Contohnya, saat mahasiswa mencentang checklist selesai, yang dikirim ke backend hanya statusnya saja tanpa perlu mengirim ulang judul atau deskripsi tugas.
3. **Penghapusan Permanen (DELETE):** Fungsi ini menjalankan perintah prisma.task.delete untuk menghapus data tugas dari database PostgreSQL secara permanen jika mahasiswa merasa tugas tersebut sudah tidak diperlukan atau salah input.

9. Manajemen Data Pengguna

api/src/app/api/users/route.ts

```
import { NextRequest, NextResponse } from "next/server";
import prisma from "@/lib/prisma";
import { getAuthUser, UNAUTHORIZED_RESPONSE } from "@/lib/auth";
import { Prisma } from "@/generated/prisma";

export async function GET(request: NextRequest) {
    try {
        // Cek user ter-autentikasi lewat token
        const authUser = getAuthUser(request)
        if (!authUser) {
            return NextResponse.json(
                { ...UNAUTHORIZED_RESPONSE },
                { status: 401 }
            )
        }

        // Ambil parameter role dari URL
        const { searchParams } = new URL(request.url)
        const role = searchParams.get('role')

        // Inisialisasi kondisi query
        let where: Prisma.UserWhereInput = {}

        // Batasi akses berdasarkan role
        if (authUser.role !== 'admin') {
            // Kalau bukan admin, kunci ke diri sendiri
```

```
            where = { id: authUser.id }
        } else {
            // Kalau admin, baru boleh filter role
            if (role) {
                where.role = role
            }
        }

        // Ambil data user tanpa password
        const users = await prisma.user.findMany({
            where,
            select: {
                id: true,
                name: true,
                email: true,
                role: true,
                npm: true,
                phone: true,
                avatar: true,
                createdAt: true
            },
            orderBy: {
                createdAt: 'desc'
            }
        })
        return NextResponse.json(
            { success: true, data: users },
            { status: 200 }
        )
    } catch (error: unknown) {
        if (error instanceof Error) {
            console.error("Get user error:", error.message)
        } else {
            console.error("Get user error:", error)
        }
        return NextResponse.json(
            { success: false, error: "Terjadi kesalahan", message: "Terjadi
kesalahan saat mengambil data user" },
            { status: 500 }
        )
    }
}
```

Deskripsi: File ini adalah endpoint untuk mengambil daftar pengguna. Di sini saya menerapkan logika Role-Based Access Control (RBAC) yang cukup ketat agar data privasi mahasiswa aman.

Alur Kerja:

1. **Pembatasan Akses (Admin & Student):** Sistem mengecek peran user yang sedang login (authUser.role):
    - **Jika Admin:** Boleh melihat semua data user dan bisa memfilter berdasarkan role (misal: mau lihat daftar dosen saja).
    - **Jika Mahasiswa:** Sistem otomatis membatasi query database (where: { id: authUser.id }). Jadi, kalau mahasiswa memanggil API ini yang keluar hanya data dirinya sendiri. Dia tidak akan bisa melihat data mahasiswa lain.
2. **Keamanan Data Sensitif:** Saya menggunakan select untuk memastikan kolom sensitif seperti password dan firebaseUid tidak pernah dikirim ke client. Yang dikirim hanya data publik seperti Nama, NPM, dan Avatar.

10. Detail & Update Profil

api/src/app/api/users/[id]/route.ts

```
import { NextRequest, NextResponse } from "next/server";
import { getAuthUser, hashPassword, UNAUTHORIZED_RESPONSE } from
'@/lib/auth'
import prisma from '@/lib/prisma'
import { Prisma } from "@/generated/prisma";

// GET user berdasarkan id
export async function GET(request: NextRequest, { params }: { params: { id:
string } }) {
    try {
        // Cek auth
        const authUser = getAuthUser(request)
        if (!authUser) {
            return NextResponse.json(
                { ...UNAUTHORIZED_RESPONSE },
                { status: 401 }
            )
        }

        const userId = parseInt(params.id)

        // SECURITY CHECK (PENTING!) kalau bukan admin DAN bukan akun
sendiri, tolak aksesnya
        if (authUser.role !== 'admin' && authUser.id !== userId) {
            return NextResponse.json(
                { success: false, error: "Forbidden", message: "Anda tidak
memiliki akses untuk melihat profil ini" },
                { status: 403 }
            )
        }
```

```
        // Ambil data user berdasarkan id
        const user = await prisma.user.findUnique({
            where: { id: userId },
            select: {
                id: true,
                name: true,
                email: true,
                role: true,
                npm: true,
                phone: true,
                avatar: true,
                createdAt: true,
                updatedAt: true,
                _count: {
                    select: {
                        tasks: true,
                        attendances: true,
                        products: true
                    }
                }
            }
        })
        if (!user) {
            return NextResponse.json(
                { success: false, error: "Not Found", message: "User tidak
ditemukan" },
                { status: 404 }
            )
        }
        return NextResponse.json(
            { success: true, data: user },
            { status: 200 }
        )
    } catch (error: unknown) {
        if (error instanceof Error) {
            console.error("Get user error:", error.message)
        } else {
            console.error("Get user error:", error)
        }
        return NextResponse.json(
            { success: false, error: "Internal Server Error", message:
"Terjadi kesalahan saat mengambil data user" },
            { status: 500 }
        )
    }
}
```

```
// UPDATE user profile
export async function PUT(request: NextRequest, { params }: { params: { id:
string } }) {
    try {
        const authUser = getAuthUser(request)
        if (!authUser) {
            return NextResponse.json(
                { ...UNAUTHORIZED_RESPONSE },
                { status: 401 }
            )
        }

        const userId = parseInt(params.id)

        // User hanya bisa update profile sendiri (kecuali admin)
        if (authUser.id !== userId && authUser.role !== 'admin' ) {
            return NextResponse.json(
                { success: false, error: "Forbidden", message: "Anda tidak
bisa mengubah profile user lain" },
                { status: 403 }
            )
        }

        const body = await request.json();
        const { name, phone, avatar, password } = body

        // Siapkan data update
        const updateData: Prisma.UserUpdateInput = {}

        if (name) updateData.name = name
        if (phone !== undefined ) updateData.phone = phone
        if (avatar !== undefined ) updateData.avatar = avatar
        if (password) {
            updateData.password = await hashPassword(password)
        }

        // Update user
        const updatedUser = await prisma.user.update({
            where: { id: userId },
            data: updateData,
            select: {
                id: true,
                name: true,
                email: true,
                role: true,
                npm: true,
                phone: true,
                avatar: true,
```

```
                createdAt: true,
                updatedAt: true
            }
        })
        return NextResponse.json(
            { success: true, message: "Profile berhasil diupdate", data:
updatedUser },
            { status: 200 }
        )

    } catch (error: unknown) {
        if (error instanceof Error) {
            console.error('Update user error:', error.message)
        } else {
            console.error('Update user error:', error)
        }
        return NextResponse.json(
            { success: false, error: "Internal Error", message: "Terjadi
kesalahan saat update profile" },
            { status: 500 }
        )
    }
}
```

Deskripsi: Endpoint ini menangani dua fungsi: GET (Melihat detail profil) dan PUT (Mengupdate data diri).

Alur Kerja:

1. **Proteksi Privasi (Forbidden Access):** Di fungsi GET dan PUT, saya memasang Pagar Keamanan if (authUser.role !== 'admin' && authUser.id !== userId) Artinya, profil si A hanya bisa dibuka/diedit oleh si A sendiri atau Admin. Mahasiswa B tidak akan bisa mengintip profil Mahasiswa A meskipun dia mencoba menembak API-nya lewat Postman/ApiDog.
2. **Rekap Data Otomatis (Dashboard Stats):** Pada fungsi GET, saya menggunakan fitur _count dari Prisma. Tujuannya agar saat profil dimuat, sistem sekalian memberitahu jumlah tugas (tasks), jumlah kehadiran (attendances), dan produk yang dijual. Ini efisien karena Frontend tidak perlu melakukan 3x request terpisah, cukup 1x request ke endpoint ini.
3. **Update Profil Aman (PUT):** Fitur ini memungkinkan user mengganti Nama, No HP, Avatar, atau Password. Khusus untuk password, sebelum disimpan ke database, sistem akan melakukan Hashing terlebih dahulu menggunakan fungsi hashPassword agar tidak tersimpan sebagai teks biasa (plain text).