LAPORAN INVENTORY PRODUCT, MENGGUNAKAN, NEXT JS, POSTGRESQL, PRISMA

ANGGOTA KELOMPOK:

1. HILGA ADITIYA PRATAMA (23104410012)
2.DAVA RIZKY ZANUAR (23104410024)
3.FAGA ARTEO PRANATA (231004410057)
4.DAVID IKSAN ZANUAR (23104410025)
5.RIZAL TEGAR W.S (23104410088)

PRODI TEKNIK INFORMATIKA
FAKULTAS TEKNOLOGI INFORMASI
UNIVERSITAS ISLAM BALITAR
2025

KATA PENGANTAR

Puji syukur penulis panjatkan ke hadirat Tuhan Yang Maha Esa karena atas rahmat dan karunia-Nya laporan berjudul “Membuat API Menggunakan Next.js, PostgreSQL, dan Penerapan Prisma” ini dapat disusun dan diselesaikan dengan baik. Laporan ini bertujuan untuk memberikan penjelasan yang lebih mendalam mengenai proses perancangan dan pembangunan API menggunakan framework **Next.js**, pengelolaan basis data dengan **PostgreSQL**, serta implementasi **Prisma** sebagai ORM untuk mempermudah interaksi antara aplikasi dan database.

Penyusunan laporan ini tidak terlepas dari dukungan berbagai pihak yang telah memberikan bantuan, bimbingan, serta masukan berharga, baik secara langsung maupun tidak langsung. Oleh karena itu, penulis mengucapkan terima kasih yang sebesar-besarnya kepada semua pihak yang telah berkontribusi dalam proses penyelesaian laporan ini.

Penulis menyadari bahwa laporan ini masih memiliki keterbatasan. Oleh sebab itu, penulis sangat mengharapkan kritik dan saran yang membangun demi perbaikan pada karya ilmiah di masa mendatang.

Akhir kata, penulis berharap laporan ini dapat memberikan manfaat bagi para pembaca, khususnya bagi pihak yang ingin memahami proses pembuatan API modern menggunakan **Next.js**, pengelolaan data dengan **PostgreSQL**, serta penerapan **Prisma** dalam pengembangan aplikasi web.

Hormat kami

(Tim Penyusun)

I. PENDAHULUAN

1.1. Latar Belakang

Di era digital saat ini, kebutuhan akan aplikasi yang cepat, efisien, dan mudah dikembangkan semakin meningkat. Hampir setiap sistem modern membutuhkan API (Application Programming Interface) sebagai penghubung antara aplikasi dengan database atau layanan lain. API berfungsi sebagai jembatan yang memungkinkan berbagai sistem saling berkomunikasi dengan aman dan terstruktur.

Untuk memenuhi kebutuhan tersebut, diperlukan teknologi yang mampu memberikan proses pengembangan yang fleksibel, mudah dipahami, serta memiliki performa yang baik. **Next.js** menjadi salah satu framework JavaScript yang banyak digunakan karena menyediakan fitur server-side dan full-stack development dalam satu lingkungan. Dengan Next.js, pengembang dapat membangun API secara langsung tanpa memerlukan backend terpisah, sehingga proses pengembangan menjadi lebih efisien.

Selain itu, pengelolaan data menjadi bagian penting dalam aplikasi. **PostgreSQL** dipilih karena merupakan database yang stabil, aman, dan mendukung berbagai kebutuhan aplikasi skala kecil hingga besar. Agar proses pengelolaan database lebih mudah, digunakan **Prisma** sebagai ORM (Object Relational Mapping). Prisma mempermudah pengembang dalam membuat, membaca, memperbarui, dan menghapus data tanpa harus menulis query SQL secara manual.

Melalui kombinasi Next.js, PostgreSQL, dan Prisma, pengembang dapat membangun API yang modern, cepat, dan mudah dipelihara. Oleh karena itu, laporan ini disusun untuk memberikan pemahaman mengenai bagaimana proses pembuatan API dilakukan menggunakan ketiga teknologi tersebut, mulai dari konfigurasi awal, pembuatan struktur database, hingga implementasi fungsi API dalam aplikasi.

1.2 Rumusan Masalah

1. Bagaimana cara membuat API menggunakan Next.js?
2. Bagaimana mengatur dan menggunakan database PostgreSQL untuk API?
3. Bagaimana Prisma membantu menghubungkan API dengan database?
4. Bagaimana menggabungkan Next.js, PostgreSQL, dan Prisma agar API bisa berjalan dengan baik?

1.3 Tujuan

1. Membuat API dengan Next.js
   Belajar membuat endpoint sederhana (GET, POST, PUT, DELETE).
2. Menggunakan PostgreSQL untuk menyimpan data
   Menyiapkan database dan menghubungkannya ke project.
3. Memahami Prisma sebagai penghubung API–database
   Membuat model, migration, dan query data dengan mudah.
4. Menggabungkan Next.js + PostgreSQL + Prisma
   Membuat A PI yang bisa mengambil, menambah, mengubah, dan menghapus data dari database.

II. LANDASAN TEORI

2.1 Pengertian Membuat API menggunakan Next.js

API di Next.js adalah cara kita membuat "jembatan" yang menghubungkan frontend dengan data di server. Dengan Next.js, kita bisa membuat file khusus di folder app/api/ yang otomatis menjadi *endpoint*. Endpoint itu bisa menerima permintaan (GET, POST, PUT, DELETE) dan mengembalikan data. Next.js memudahkan kita bikin API tanpa harus pakai server terpisah.

2.2 Pengertian PostgreSQL

PostgreSQL adalah **database** yang menyimpan data secara terstruktur, seperti data pengguna, produk, transaksi, dll. Kelebihannya: kuat, aman, dan banyak digunakan untuk aplikasi skala besar. PostgreSQL tempat menyimpan data yang nanti diakses lewat API.

2.3 Pengertian PRISMA

Prisma adalah ORM semacam alat yang membantu kita berkomunikasi dengan database PostgreSQL menggunakan kode JavaScript/TypeScript, bukan SQL manual. Dengan Prisma, kita bisa membuat model data, migration, dan melakukan CRUD dengan mudah. Prisma menjadi penghubung antara API Next.js dan database PostgreSQL.

2.4 Pengertian Security Layer

Security layer adalah **lapisan keamanan** yang dipasang pada API agar data tetap aman dari penyalahgunaan.
Contohnya:

A. Validasi input
B. Proteksi dari SQL Injection
C. Penggunaan token (JWT) untuk autentikasi
D. Pembatasan akses (authorization)

Security layer menjaga API supaya tidak mudah diretas atau disalahgunakan.

2.5 Pengertian Postman

Postman adalah sebuah aplikasi yang digunakan untuk menguji, mengirim, dan mengelola API dengan cara yang mudah. Dengan Postman, kamu bisa mencoba sebuah API tanpa harus membuat program dulu. Fungsinya Menguji apakah API berjalan dengan benar, Mengirim data ke server, memudahkan developer bekerja dengan API tanpa ribet

III. IMPLEMENTASI SISTEM

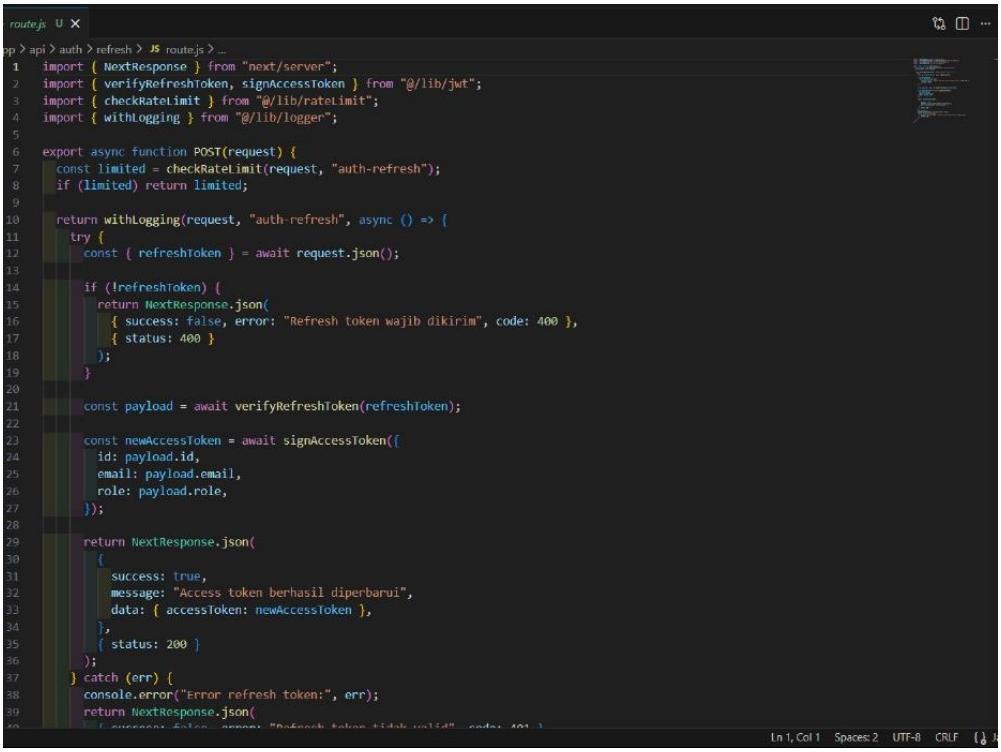
3.1 Login

route.js U

app > api > auth > login > route.js > ...

```
import { NextResponse } from "next/server";
import { prisma } from "@/lib/prisma";
import bcrypt from "bcryptjs";
import { signAccessToken, signRefreshToken } from "@/lib/jwt";
import { checkRateLimit } from "@/lib/rateLimit";
import { withLogging } from "@/lib/logger";

export async function POST(request) {
  const limited = checkRateLimit(request, "auth-login");
  if (limited) return limited;

  return withLogging(request, "auth-login", async () => {
    try {
      const { email, password } = await request.json();

      if (!email || !password) {
        return NextResponse.json(
          { success: false, error: "Email & password wajib diisi", code: 400 },
          { status: 400 }
        );
      }

      const user = await prisma.user.findUnique({ where: { email } });
      if (!user) {
        return NextResponse.json(
          { success: false, error: "Invalid credentials", code: 401 },
          { status: 401 }
        );
      }

      const match = await bcrypt.compare(password, user.password);
      if (!match) {
        return NextResponse.json(
          { success: false, error: "Invalid credentials", code: 401 },
          { status: 401 }
        );
      }

      const accessToken = await signAccessToken(user);
```

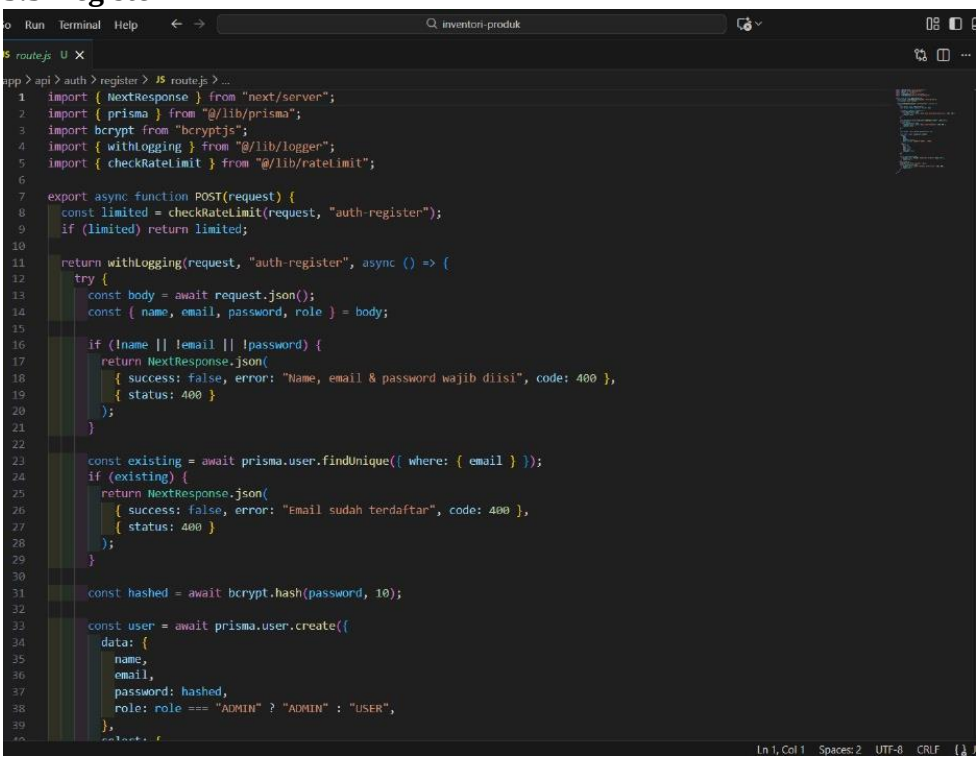
3.2 Refresh Token



```
route.js U
app > api > auth > refresh > JS route.js > ...
import { NextResponse } from "next/server";
import { verifyRefreshToken, signAccessToken } from "@/lib/jwt";
import { checkRateLimit } from "@/lib/rateLimit";
import { withLogging } from "@/lib/logger";

export async function POST(request) {
  const limited = checkRateLimit(request, "auth-refresh");
  if (limited) return limited;

  return withLogging(request, "auth-refresh", async () => {
    try {
      const { refreshToken } = await request.json();

      if (!refreshToken) {
        return NextResponse.json(
          { success: false, error: "Refresh token wajib dikirim", code: 400 },
          { status: 400 }
        );
      }

      const payload = await verifyRefreshToken(refreshToken);

      const newAccessToken = await signAccessToken({
        id: payload.id,
        email: payload.email,
        role: payload.role,
      });

      return NextResponse.json(
        {
          success: true,
          message: "Access token berhasil diperbarui",
          data: { accessToken: newAccessToken },
        },
        { status: 200 }
      );
    } catch (err) {
      console.error("Error refresh token:", err);
      return NextResponse.json(
```

Ln 1, Col 1 Spaces: 2 UTF-8 CRLF

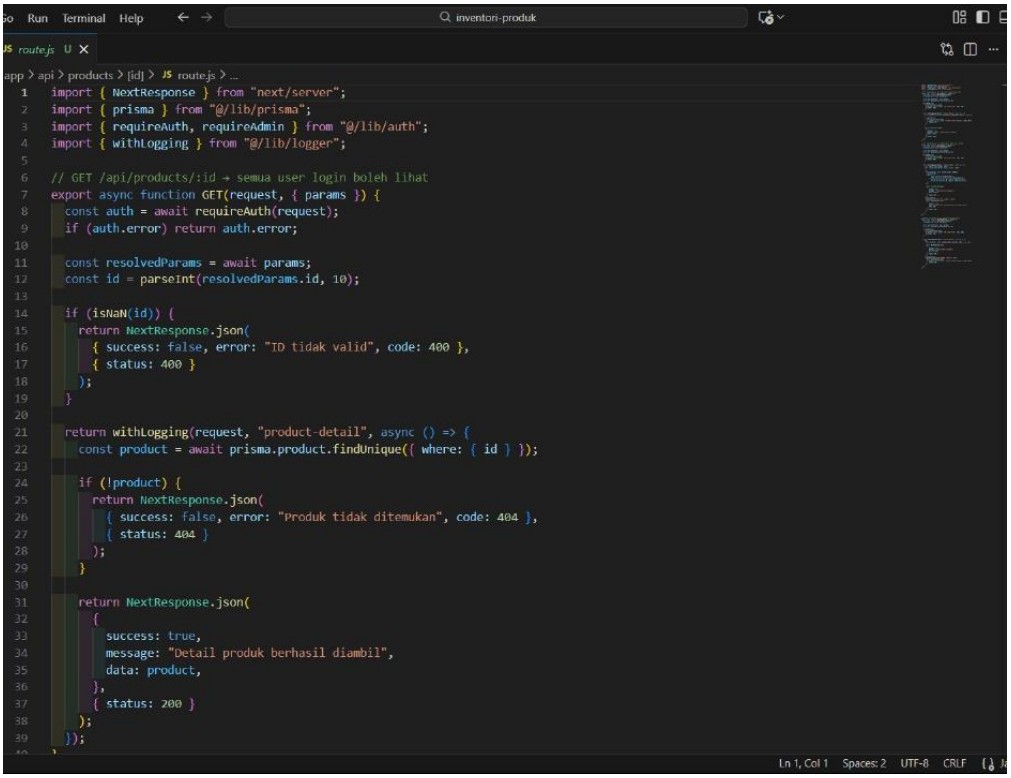
3.3 Register



```
o Run Terminal Help
inventori-produk
route.js U
app > api > auth > register > JS route.js > ...
import { NextResponse } from "next/server";
import { prisma } from "@/lib/prisma";
import bcrypt from "bcryptjs";
import { withLogging } from "@/lib/logger";
import { checkRateLimit } from "@/lib/rateLimit";

export async function POST(request) {
  const limited = checkRateLimit(request, "auth-register");
  if (limited) return limited;

  return withLogging(request, "auth-register", async () => {
    try {
      const body = await request.json();
      const { name, email, password, role } = body;

      if (!name || !email || !password) {
        return NextResponse.json(
          { success: false, error: "Name, email & password wajib diisi", code: 400 },
          { status: 400 }
        );
      }

      const existing = await prisma.user.findUnique({ where: { email } });
      if (existing) {
        return NextResponse.json(
          { success: false, error: "Email sudah terdaftar", code: 400 },
          { status: 400 }
        );
      }

      const hashed = await bcrypt.hash(password, 10);

      const user = await prisma.user.create({
        data: {
          name,
          email,
          password: hashed,
          role: role === "ADMIN" ? "ADMIN" : "USER",
        },
```

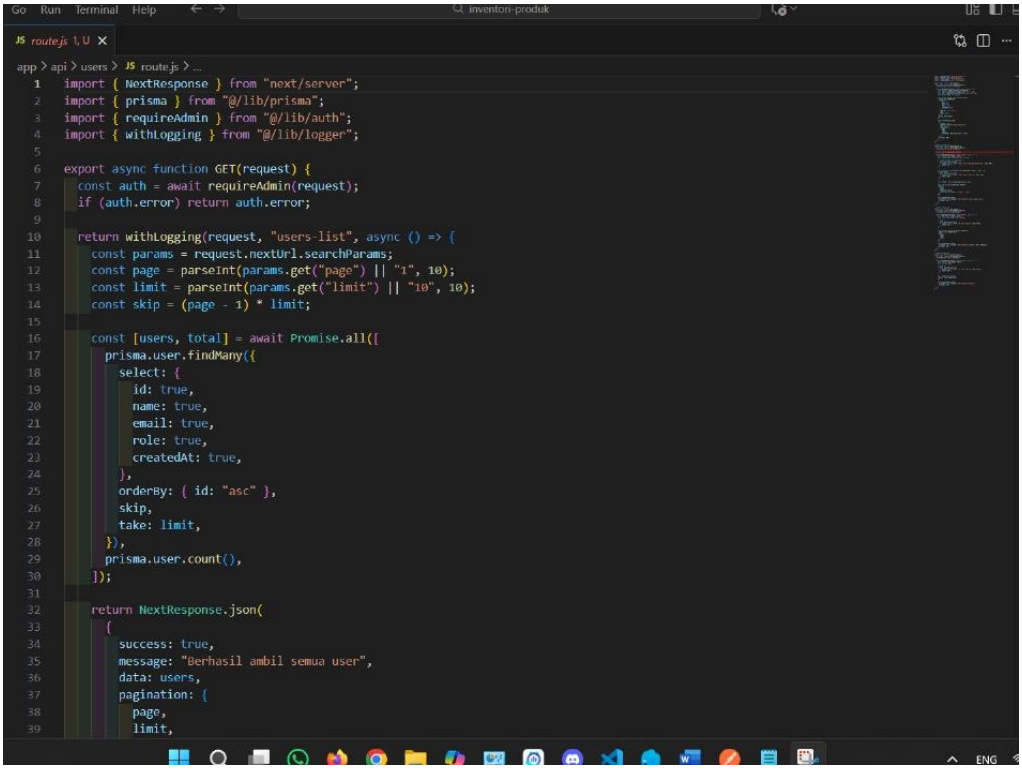
Ln 1, Col 1 Spaces: 2 UTF-8 CRLF

3.4 Product



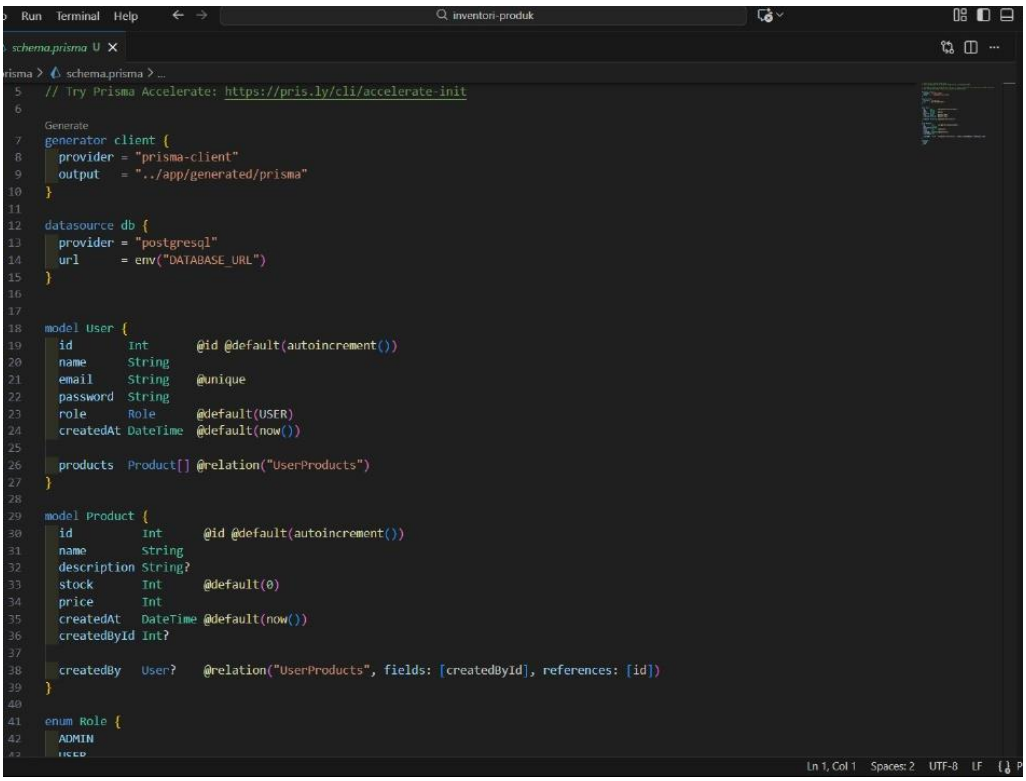
```
app > api > products > [id] > route.js > ...
import { NextResponse } from "next/server";
import { prisma } from "@/lib/prisma";
import { requireAuth, requireAdmin } from "@/lib/auth";
import { withLogging } from "@/lib/logger";

// GET /api/products/:id → semua user login boleh lihat
export async function GET(request, { params }) {
  const auth = await requireAuth(request);
  if (auth.error) return auth.error;

  const resolvedParams = await params;
  const id = parseInt(resolvedParams.id, 10);

  if (isNaN(id)) {
    return NextResponse.json(
      { success: false, error: "ID tidak valid", code: 400 },
      { status: 400 }
    );
  }

  return withLogging(request, "product-detail", async () => {
    const product = await prisma.product.findUnique({ where: { id } });

    if (!product) {
      return NextResponse.json(
        { success: false, error: "Produk tidak ditemukan", code: 404 },
        { status: 404 }
      );
    }

    return NextResponse.json(
      {
        success: true,
        message: "Detail produk berhasil diambil",
        data: product,
      },
      { status: 200 }
    );
  });
}
```

3.5 User



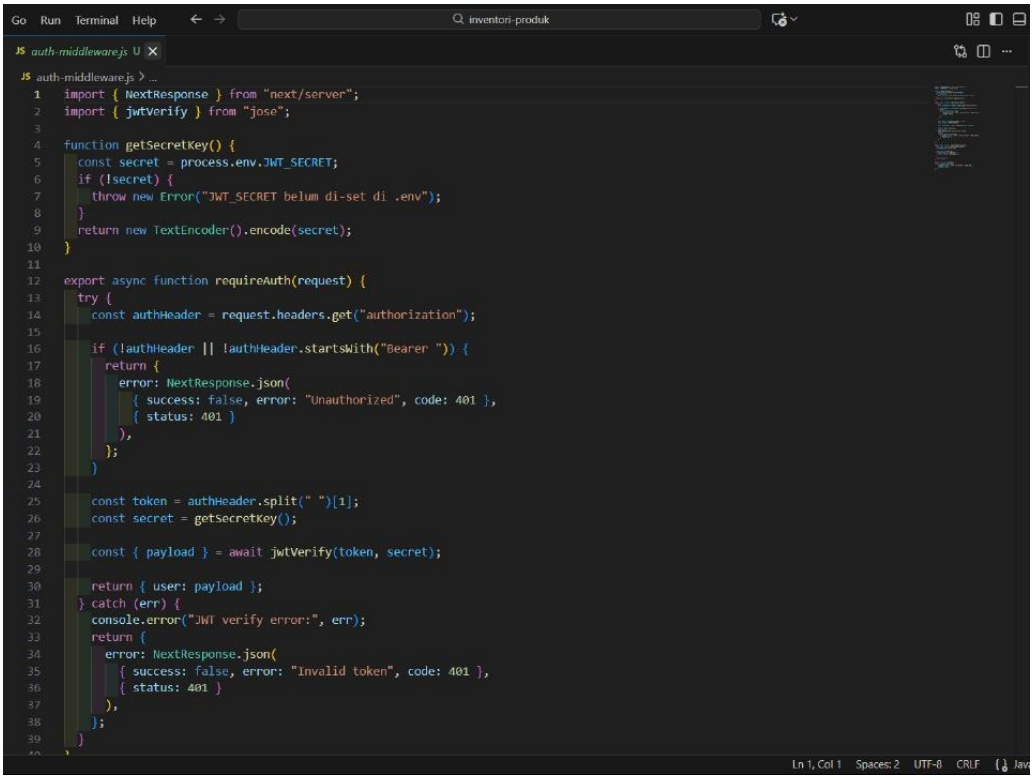
```
app > api > users > route.js > ...
import { NextResponse } from "next/server";
import { prisma } from "@/lib/prisma";
import { requireAdmin } from "@/lib/auth";
import { withLogging } from "@/lib/logger";

export async function GET(request) {
  const auth = await requireAdmin(request);
  if (auth.error) return auth.error;

  return withLogging(request, "users-list", async () => {
    const params = request.nextUrl.searchParams;
    const page = parseInt(params.get("page") || "1", 10);
    const limit = parseInt(params.get("limit") || "10", 10);
    const skip = (page - 1) * limit;

    const [users, total] = await Promise.all([
      prisma.user.findMany({
        select: {
          id: true,
          name: true,
          email: true,
          role: true,
          createdAt: true,
        },
        orderBy: { id: "asc" },
        skip,
        take: limit,
      }),
      prisma.user.count(),
    ]);

    return NextResponse.json(
      {
        success: true,
        message: "Berhasil ambil semua user",
        data: users,
        pagination: {
          page,
          limit,
```

3.6 Scema Prisma



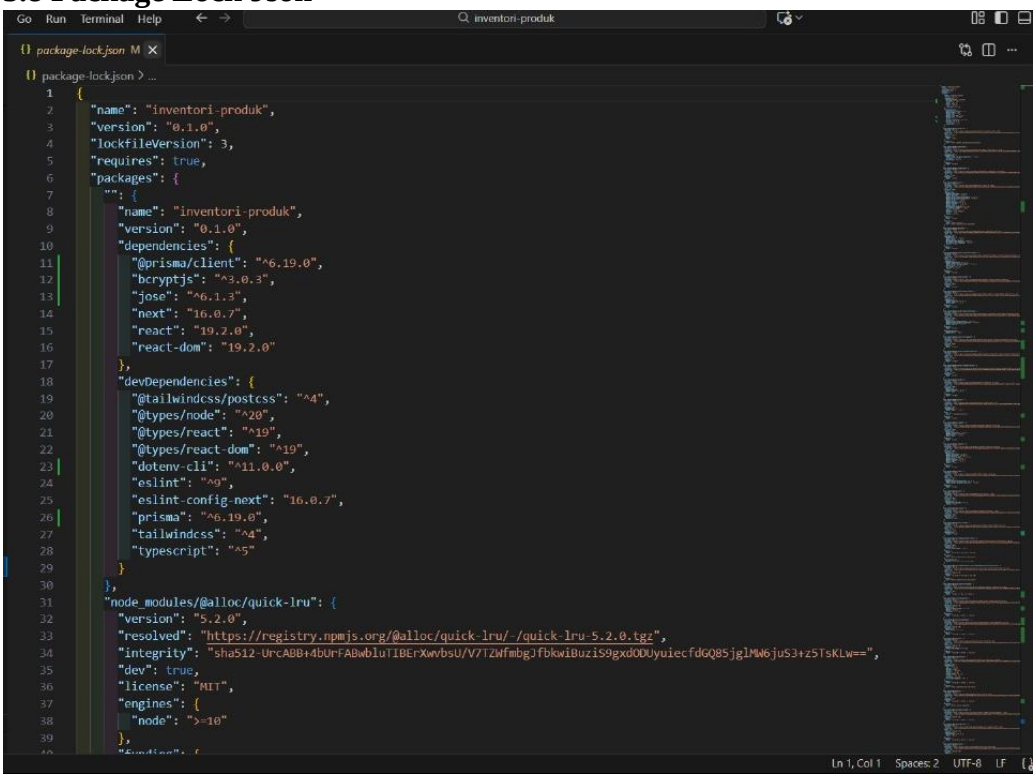
```
// Try Prisma Accelerate: https://pris.ly/cli/accelerate-init

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

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


model User {
  id        Int       @id @default(autoincrement())
  name      String
  email     String    @unique
  password  String
  role      Role      @default(USER)
  createdAt DateTime  @default(now())

  products  Product[] @relation("UserProducts")
}

model Product {
  id          Int      @id @default(autoincrement())
  name        String
  description String?
  stock       Int      @default(0)
  price       Int
  createdAt   DateTime @default(now())
  createdById Int?

  createdBy   User?    @relation("UserProducts", fields: [createdById], references: [id])
}

enum Role {
  ADMIN
```

3.7 Auth Middleware

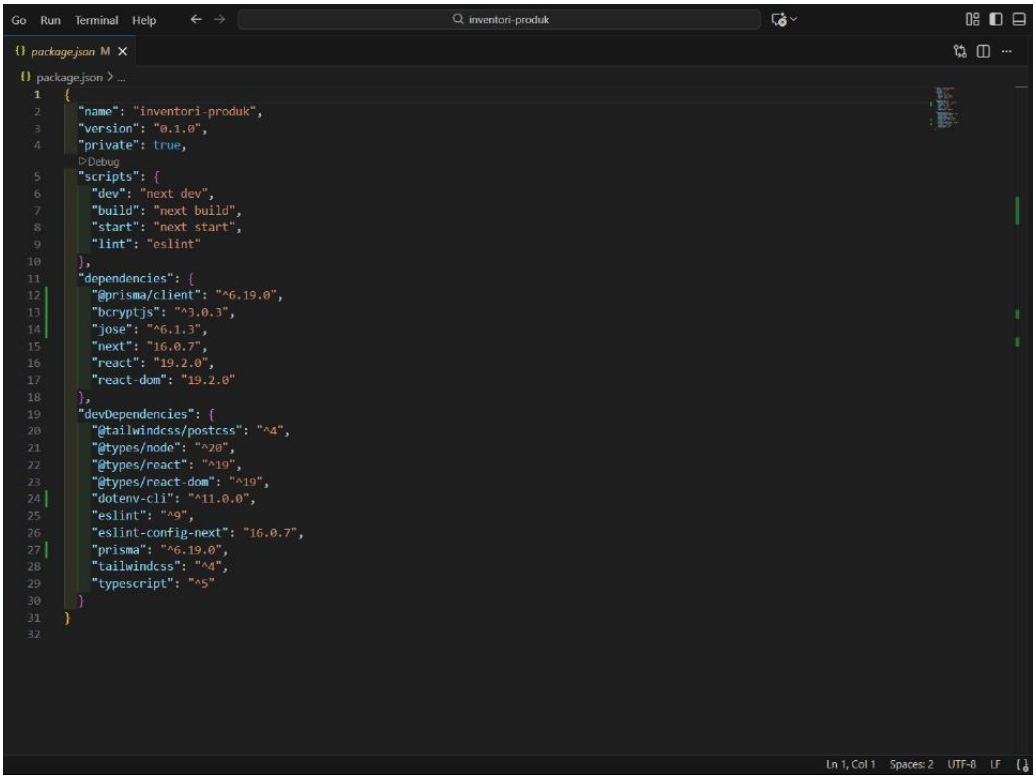


```
import { NextResponse } from "next/server";
import { jwtVerify } from "jose";

function getSecretKey() {
  const secret = process.env.JWT_SECRET;
  if (!secret) {
    throw new Error("JWT_SECRET belum di-set di .env");
  }
  return new TextEncoder().encode(secret);
}

export async function requireAuth(request) {
  try {
    const authHeader = request.headers.get("authorization");

    if (!authHeader || !authHeader.startsWith("Bearer ")) {
      return {
        error: NextResponse.json(
          { success: false, error: "Unauthorized", code: 401 },
          { status: 401 }
        ),
      };
    }

    const token = authHeader.split(" ")[1];
    const secret = getSecretKey();

    const { payload } = await jwtVerify(token, secret);

    return { user: payload };
  } catch (err) {
    console.error("JWT verify error:", err);
    return {
      error: NextResponse.json(
        { success: false, error: "Invalid token", code: 401 },
        { status: 401 }
      ),
    };
  }
}
```

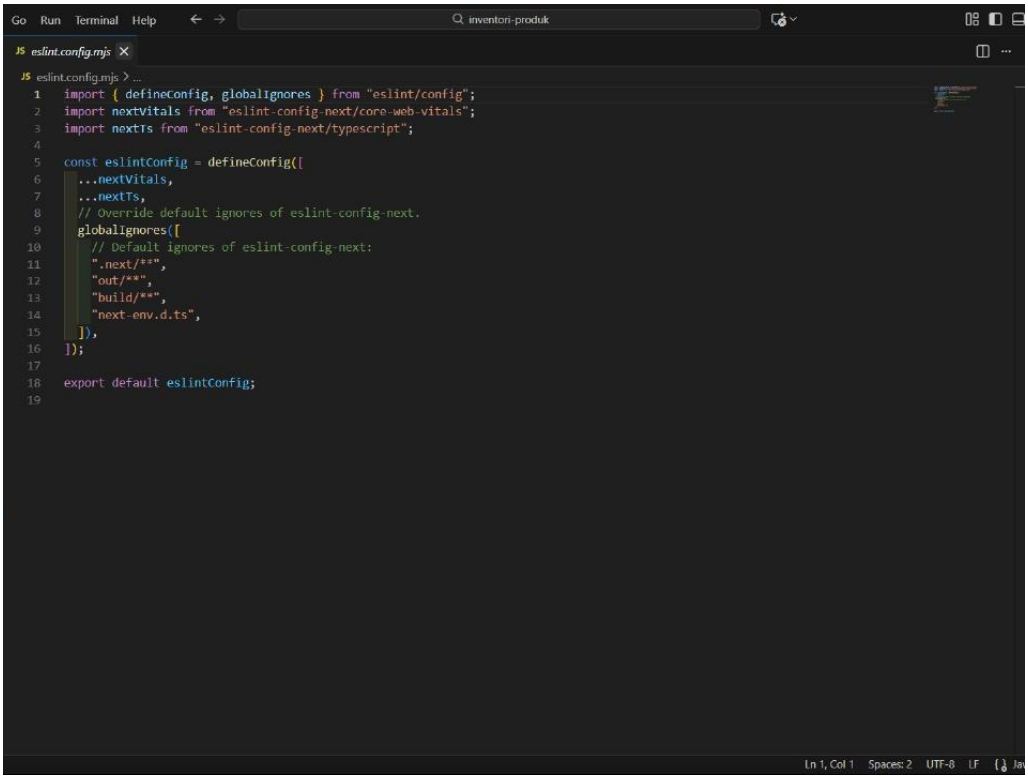
3.8 Package Lock Json



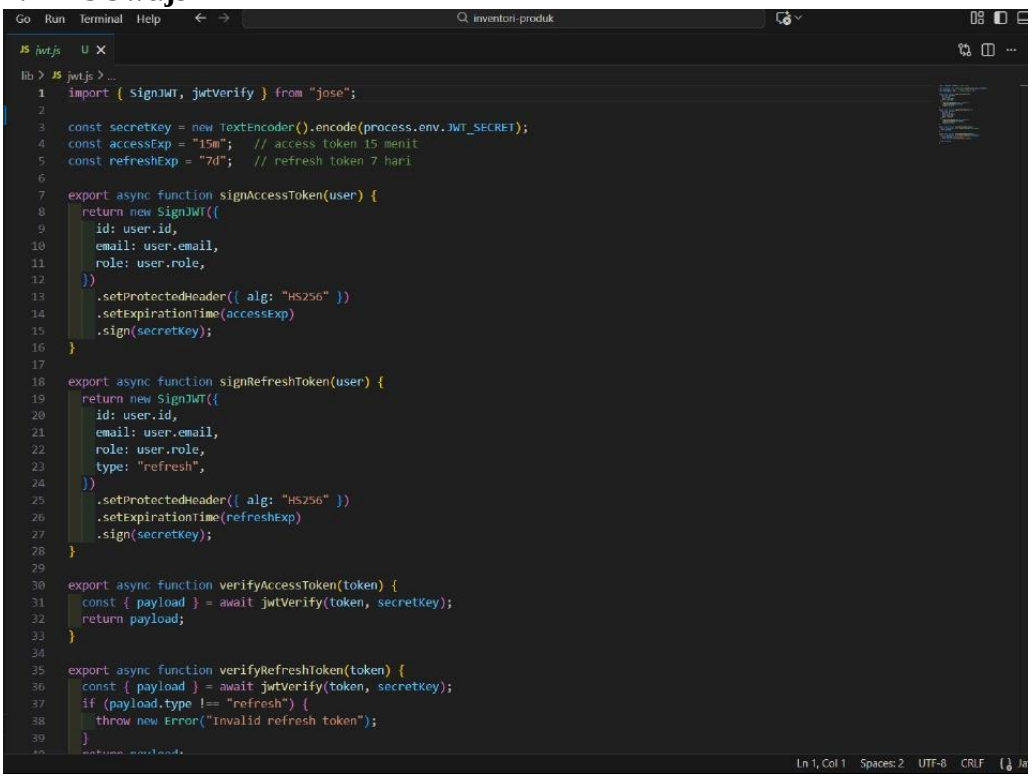
3.9 Package Json



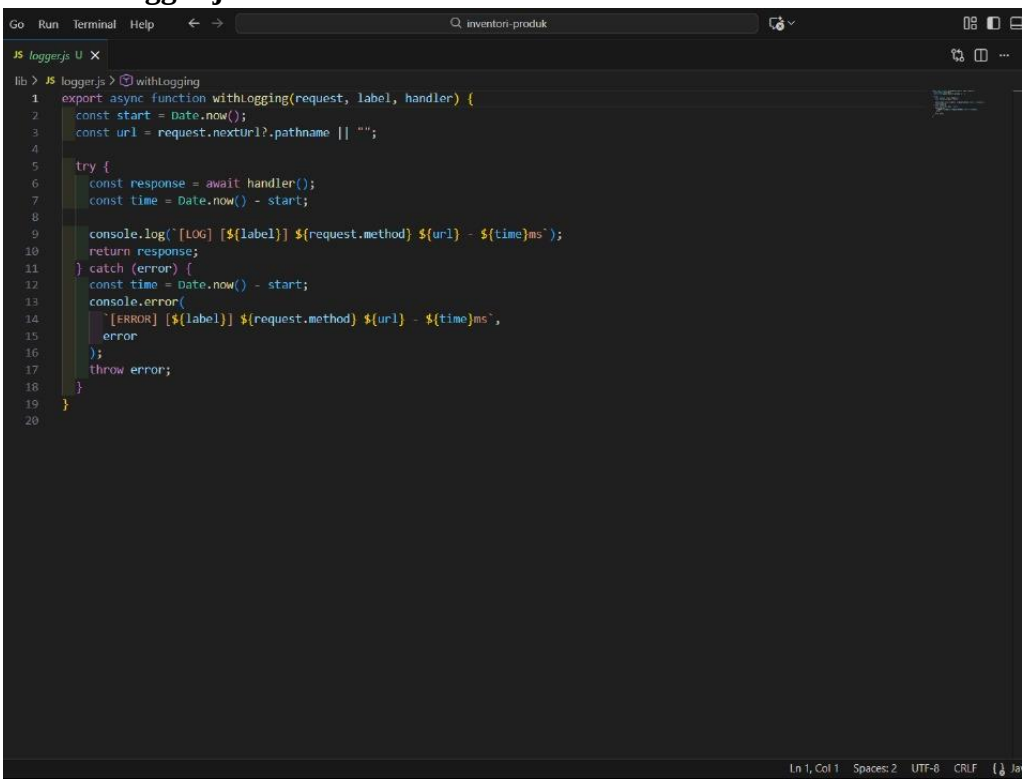
4.0 Eslint Config. mjs



4.1 Lib Jwt.js

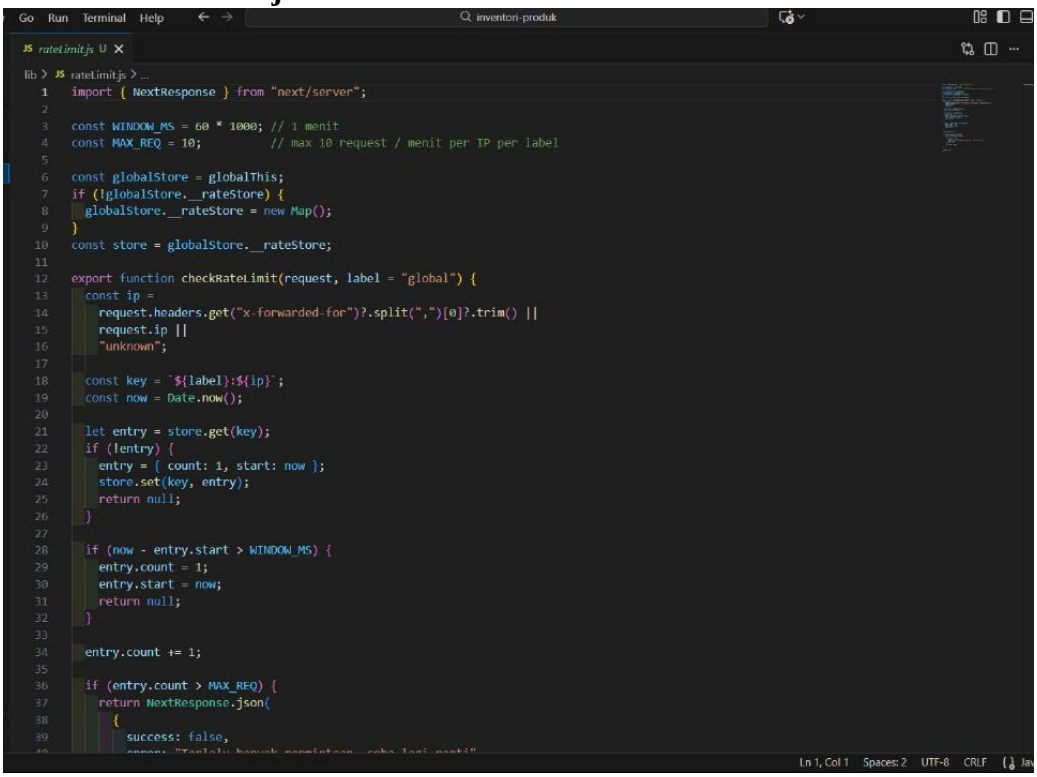


4.2 Lib Logger.js



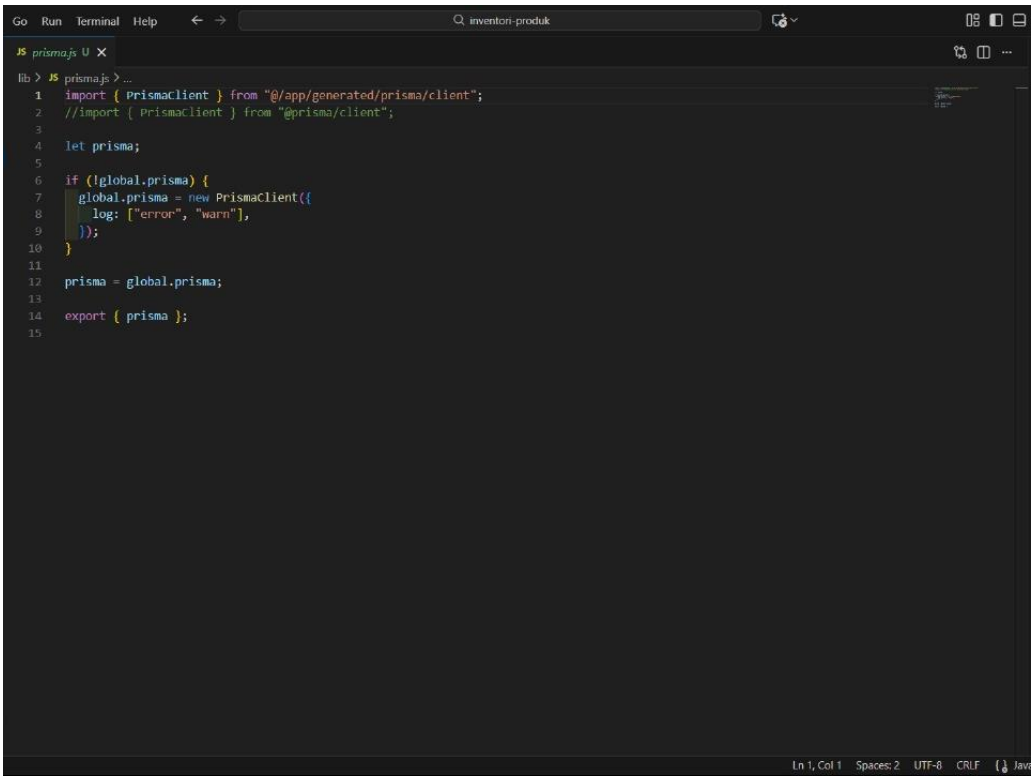
```
export async function withLogging(request, label, handler) {
  const start = Date.now();
  const url = request.nextUrl?.pathname || "";

  try {
    const response = await handler();
    const time = Date.now() - start;

    console.log(`[LOG] [${label}] ${request.method} ${url} - ${time}ms`);
    return response;
  } catch (error) {
    const time = Date.now() - start;
    console.error(
      `[ERROR] [${label}] ${request.method} ${url} - ${time}ms`,
      error
    );
    throw error;
  }
}
```

4.3 Lib Rate. Limit.js



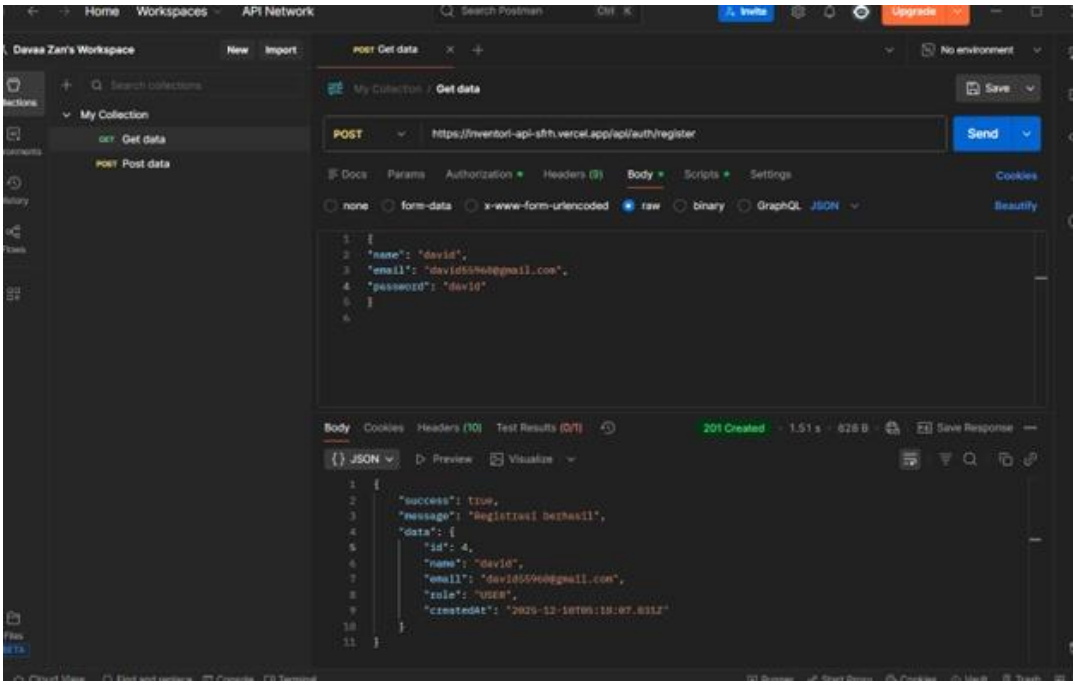
```
import { NextResponse } from "next/server";

const WINDOW_MS = 60 * 1000; // 1 menit
const MAX_REQ = 10;          // max 10 request / menit per IP per label

const globalStore = globalThis;
if (!globalStore.__rateStore) {
  globalStore.__rateStore = new Map();
}
const store = globalStore.__rateStore;

export function checkRateLimit(request, label = "global") {
  const ip =
    request.headers.get("x-forwarded-for")?.split(",")[0]?.trim() ||
    request.ip ||
    "unknown";

  const key = `${label}:${ip}`;
  const now = Date.now();

  let entry = store.get(key);
  if (!entry) {
    entry = { count: 1, start: now };
    store.set(key, entry);
    return null;
  }

  if (now - entry.start > WINDOW_MS) {
    entry.count = 1;
    entry.start = now;
    return null;
  }

  entry.count += 1;

  if (entry.count > MAX_REQ) {
    return NextResponse.json(
      {
        success: false,
```

4.4 Lib Prisma.js

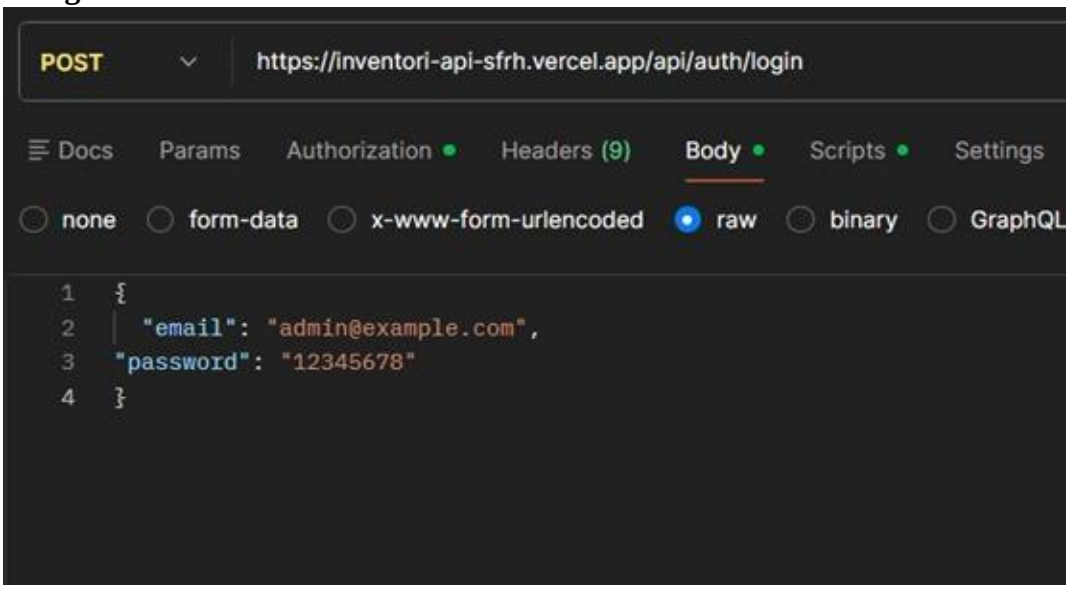


4.5 Test Postman

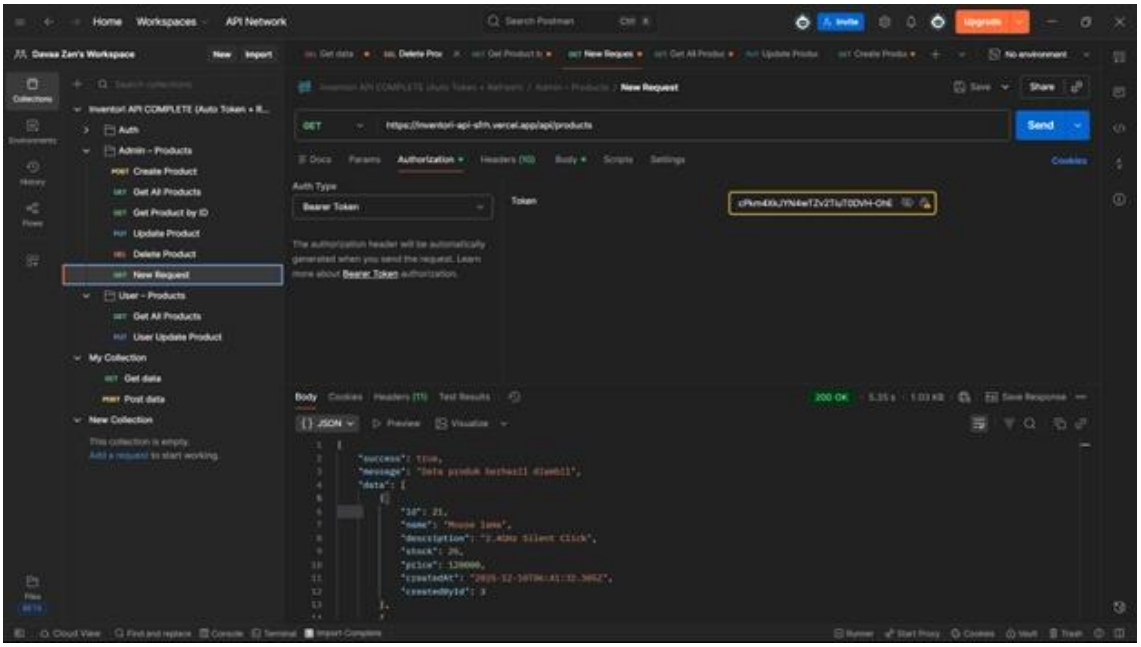
1 Register Postman



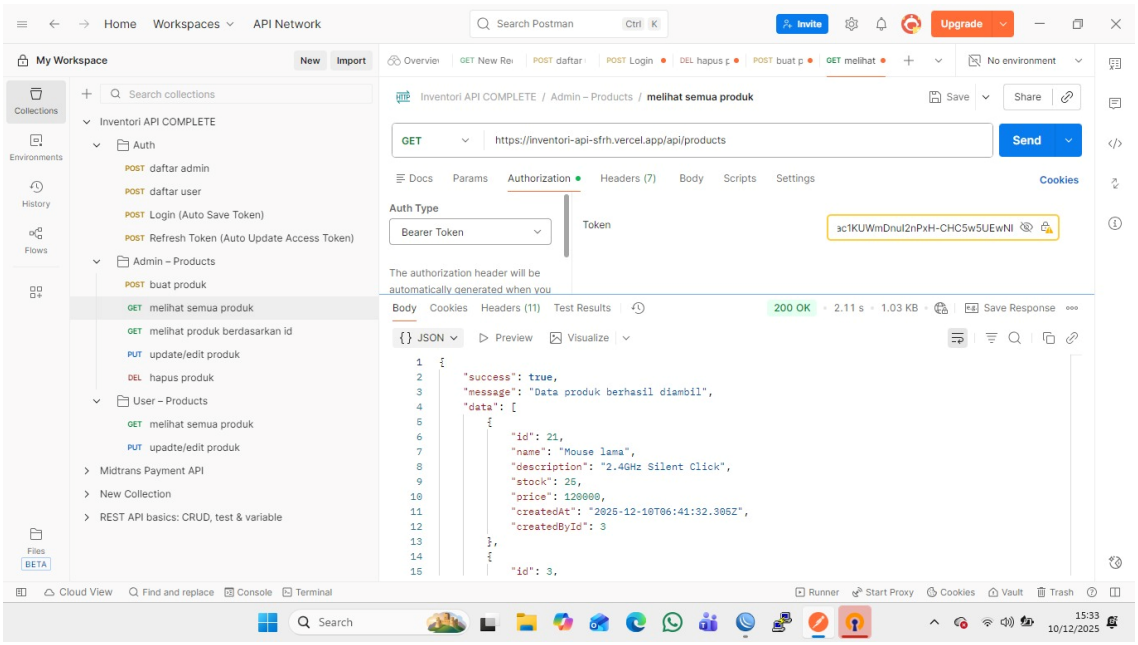
2 Login



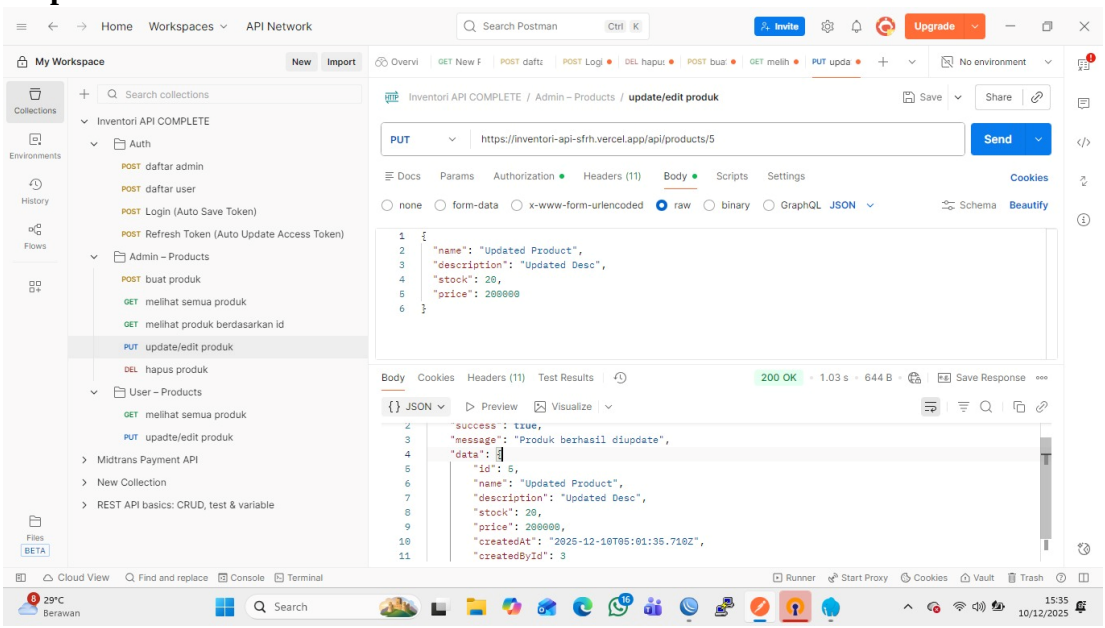
3 Accest Token



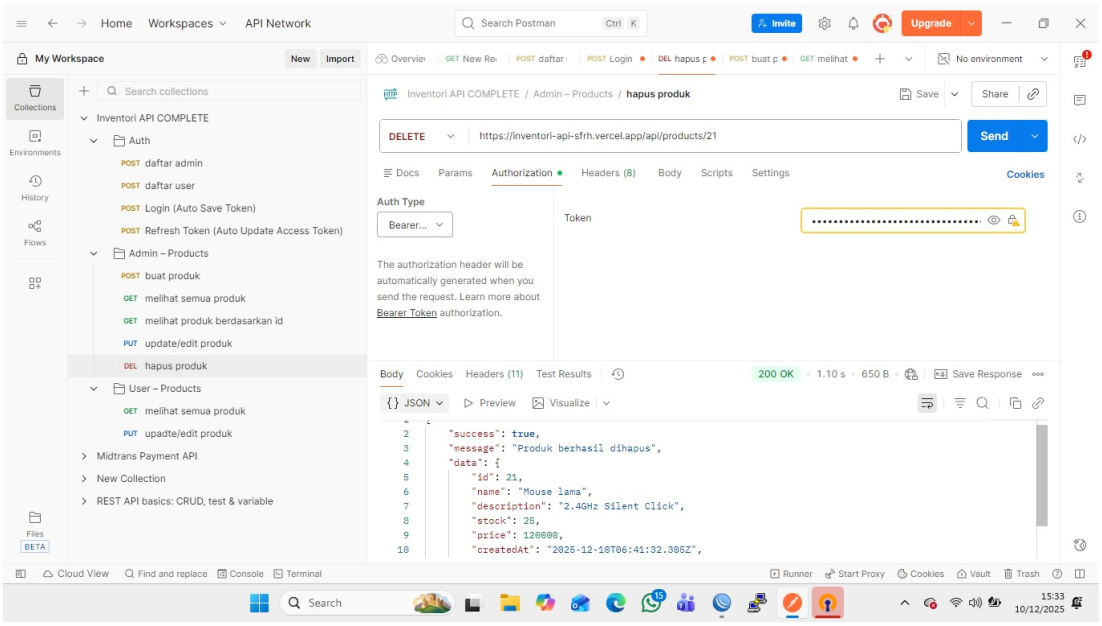
4 View all product



5 Update Product



6 Delete Product



lV. PENUTUP

5.1 KESIMPULAN

Pembuatan API menggunakan Next.js, PostgreSQL, dan Prisma memberikan cara kerja yang modern, cepat, dan efisien dalam membangun aplikasi berbasis web. Next.js mempermudah pembuatan endpoint API tanpa perlu server terpisah, PostgreSQL berperan sebagai tempat penyimpanan data yang kuat dan stabil, sementara Prisma menjadi alat penghubung yang memudahkan proses pengolahan data di database.

Dengan menambahkan security layer, API menjadi lebih aman dari serangan maupun penyalahgunaan, sehingga data dapat terjaga dengan baik. Secara keseluruhan, kombinasi teknologi ini mampu menghasilkan API yang stabil, aman, mudah dikembangkan, dan cocok untuk kebutuhan aplikasi masa kini**.**

5.2 SARAN

1. **Gunakan struktur project yang teratur dan konsisten**
   Dengan menata folder pada Next.js, model Prisma, dan file konfigurasi secara rapi, proses pengembangan akan jadi lebih mudah. Developer lain pun akan lebih cepat memahami alur kerja API.
2. **Selalu lakukan update pada Next.js, PostgreSQL, dan Prisma ketika versi stabil terbaru tersedia**
   Pembaruan versi biasanya membawa perbaikan bug, peningkatan performa, serta fitur keamanan tambahan. Ini membantu menjaga API tetap modern dan aman digunakan.

3. **Perkuat security layer sejak awal pengembangan**
   Terapkan validasi input, hashing password, penggunaan token (JWT), proteksi terhadap SQL injection, serta middleware autentikasi dan otorisasi. Semakin kuat lapisan keamanan, semakin kecil risiko API disalahgunakan.
4. **Gunakan environment variable untuk data sensitif**
   Informasi penting seperti URL database, password, secret key, dan token harus disimpan dalam file `.env`, bukan langsung di dalam kode. Ini membantu menjaga keamanan data, terutama saat project dibagikan atau di-hosting online.
5. **Lakukan pengujian API secara rutin**
   Gunakan tools seperti Postman, Thunder Client, atau Insomnia untuk mengecek apakah semua endpoint bekerja sesuai rencana. Pengujian rutin membantu menemukan error lebih cepat sebelum masuk ke tahap produksi.
6. **Buat dokumentasi API yang jelas dan mudah diikuti**
   Dokumentasi yang baik mencakup daftar endpoint, parameter, contoh respons, dan cara penggunaan. Dokumentasi sangat membantu tim lain atau pengembang baru memahami fungsi API yang dibuat.
7. **Pertimbangkan penggunaan logging dan monitoring**
   Dengan menambah fitur log, kamu bisa memantau error atau aktivitas mencurigakan. Monitoring ini penting saat API sudah digunakan banyak orang.