# LAPORAN UAS PEMROGRAMAN API


**Disusun Oleh:**

1. Gilbert Wijaya (23104410083)
2. Nanda Nalendra Bagaskara (23104410069)
3. Sandi Widya Permana (23104410053)
4. Rafii Rahmadiansyah (23104410052)
5. Rafli Fattah Nur Ikhsan (23104410055)
6. Danang Sugeng Widagdo (23104410068)

**PROGRAM STUDI TEKNIK INFORMATIKA**

**FAKULTAS TEKNIK DAN INFORMATIKA**

**UNIVERSITAS ISLAM BALITAR**

**2025**

# BAB I

## PENDAHULUAN

### 1.1 Latar Belakang

Dalam era pengembangan aplikasi web modern, arsitektur software telah bergeser menuju pendekatan yang lebih terdistribusi, di mana backend dan frontend sering kali dipisahkan untuk meningkatkan skalabilitas dan fleksibilitas. Representational State Transfer (REST) API menjadi standar industri dalam menjembatani komunikasi data antara antarmuka pengguna dan server. Kemampuan untuk membangun REST API yang handal, cepat, dan aman merupakan kompetensi fundamental yang harus dimiliki oleh seorang pengembang perangkat lunak.

Proyek Ujian Akhir Semester (UAS) ini berfokus pada pengembangan backend aplikasi manajemen data menggunakan teknologi Next.js. Meskipun dikenal sebagai kerangka kerja frontend, Next.js memiliki kemampuan server-side yang mumpuni melalui fitur API Routes-nya. Untuk pengelolaan basis data, sistem ini menggunakan PostgreSQL yang dikenal tangguh, dipadukan dengan Prisma ORM untuk mempermudah interaksi database dan menjaga keamanan type-safety dalam kode.

Selain fungsionalitas dasar Create, Read, Update, dan Delete (CRUD), aspek keamanan menjadi prioritas utama dalam pengembangan ini. Ancaman terhadap integritas data menuntut implementasi lapisan keamanan yang ketat. Oleh karena itu, sistem ini menerapkan mekanisme autentikasi dan otorisasi berlapis, mulai dari enkripsi kata sandi (password hashing) menggunakan bcrypt, penggunaan JWT (JSON Web Token) untuk manajemen sesi stateless, hingga penerapan Middleware untuk memvalidasi akses berdasarkan peran pengguna (Role-Based Authorization).

Laporan ini akan membahas perancangan dan implementasi REST API untuk Sistem Manajemen Buku, yang bertujuan untuk menyediakan layanan backend yang aman, efisien, dan terstruktur bagi pengguna dengan peran Admin maupun User biasa.

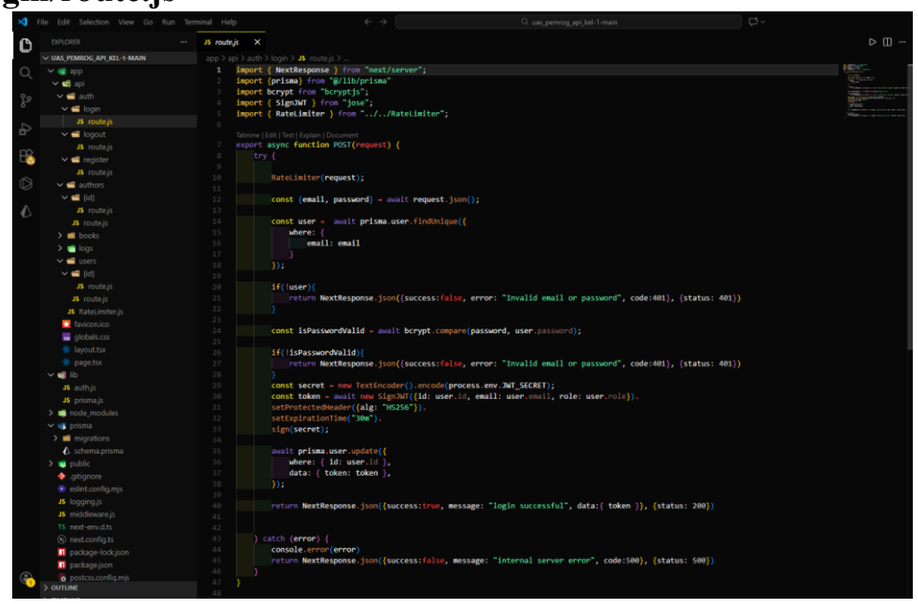
# BAB II

# PEMBAHASAN

## 2.1 Penjelasan Coding dan Codingnya

### 1. app/api/auth/…

#### A. login/route.js



```
import { NextResponse } from "next/server";
import {prisma} from "@/lib/prisma"
import bcrypt from "bcryptjs";
import { SignJWT } from "jose";
import { RateLimiter } from "../../RateLimiter";

export async function POST(request) {
    try {

        RateLimiter(request);

        const {email, password} = await request.json();

        const user =  await prisma.user.findUnique({
            where: {
                email: email
            }
        });

        if(!user){
            return NextResponse.json({success:false, error: "Invalid email or password", code:401}, {status: 401})
        }

        const isPasswordValid = await bcrypt.compare(password, user.password);

        if(!isPasswordValid){
            return NextResponse.json({success:false, error: "Invalid email or password", code:401}, {status: 401})
        }
        const secret = new TextEncoder().encode(process.env.JWT_SECRET);
        const token = await new SignJWT({id: user.id, email: user.email, role: user.role}).
        setProtectedHeader({alg: "HS256"}).
        setExpirationTime("30m").
        sign(secret);

        await prisma.user.update({
            where: { id: user.id },
            data: { token: token },
        });

        return NextResponse.json({success:true, message: "login successful", data:{ token }}, {status: 200})


    } catch (error) {
        console.error(error)
        return NextResponse.json({success:false, message: "internal server error", code:500}, {status: 500})
    }
}
```

Penjelasannya :

**1. Impor Modul (Baris 1 - 5)**

Bagian ini memanggil pustaka (library) yang diperlukan:

- **NextResponse**: Digunakan untuk mengirimkan respon HTTP (seperti status 200, 401, dll) kembali ke client.
- **prisma**: Interface untuk berinteraksi dengan database.
- **bcrypt**: Pustaka untuk keamanan password (membandingkan password yang diinput dengan yang terenkripsi di database).
- **SignJWT**: Digunakan untuk membuat token keamanan (JWT) sebagai bukti login.
- **RateLimiter**: Fungsi kustom (middleware) untuk membatasi jumlah percobaan login guna mencegah serangan *Brute Force*.

**2. Penanganan Request & Rate Limiting (Baris 7 - 10)**

- **export async function POST(request)**: Mendefinisikan bahwa rute API ini hanya menerima metode **POST**.
- **RateLimiter(request)**: Langkah pertama adalah mengecek apakah user sudah terlalu sering mencoba login. Jika ya, proses akan dihentikan di sini.

**3. Pengambilan Data & Validasi User (Baris 12 - 22)**

- **request.json()**: Mengambil data email dan password yang dikirimkan oleh user melalui body request.

- **prisma.user.findUnique**: Mencari data user di tabel database berdasarkan email yang diinput.
- **Pengecekan if (!user)**: Jika email tidak ditemukan di database, sistem langsung mengembalikan error **401 (Unauthorized)**. Pesan dibuat umum ("Invalid email or password") demi alasan keamanan.

**4. Verifikasi Password (Baris 24 - 28)**

- **bcrypt.compare**: Membandingkan password teks biasa dari user dengan password terenkripsi (hash) yang ada di database.
- **Pengecekan if (!isPasswordValid)**: Jika password salah, kembalikan error **401**.

**5. Pembuatan JWT Token (Baris 29 - 33)**

Setelah user terverifikasi, sistem membuat "tiket masuk" digital:

- **secret**: Mengambil kunci rahasia dari environment variable (JWT_SECRET).
- **SignJWT(...)**: Memasukkan data user (ID, Email, Role) ke dalam token.
- **setExpirationTime("30m")**: Token diatur agar kedaluwarsa dalam 30 menit demi keamanan.
- **sign(secret)**: Menandatangani token secara digital agar tidak bisa dipalsukan.

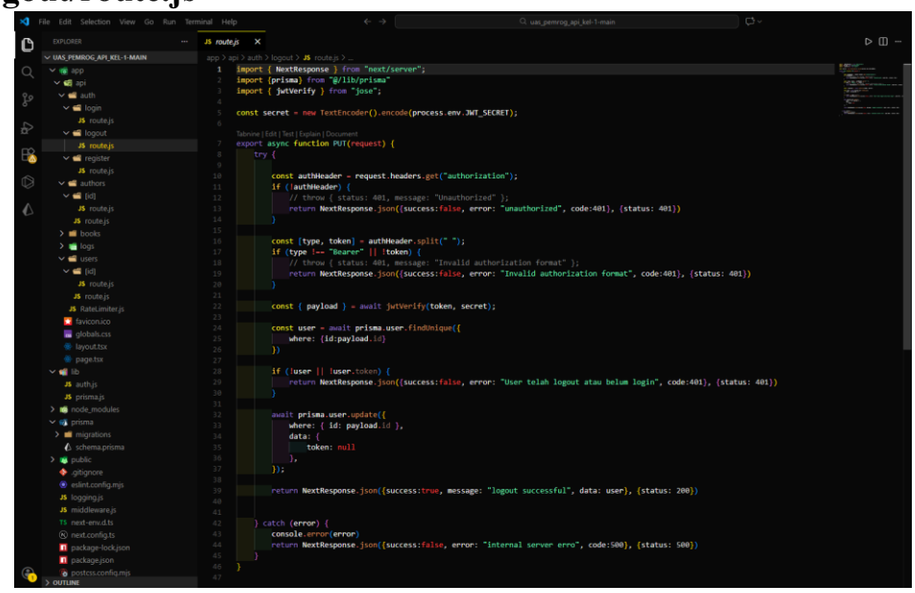
**6. Update Database & Respon Berhasil (Baris 35 - 40)**

- **prisma.user.update**: Menyimpan token yang baru dibuat ke dalam record user di database (opsional, tergantung arsitektur, biasanya untuk melacak session aktif).
- **NextResponse.json(..., {status: 200})**: Mengirimkan respon sukses ke client beserta token JWT-nya.

**7. Error Handling (Baris 43 - 46)**

- **catch (error)**: Jika terjadi kesalahan sistem (misal: database mati atau error koding), blok ini akan menangkapnya.
- **Status 500**: Mengembalikan pesan "internal server error" agar aplikasi tidak *crash* di sisi user.

## B. logout/route.js



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

const secret = new TextEncoder().encode(process.env.JWT_SECRET);

export async function PUT(request) {
    try {

        const authHeader = request.headers.get("authorization");
        if (!authHeader) {
            // throw { status: 401, message: "Unauthorized" };
            return NextResponse.json({success:false, error: "unauthorized", code:401}, {status: 401})
        }

        const [type, token] = authHeader.split(" ");
        if (type !== "Bearer" || !token) {
            // throw { status: 401, message: "Invalid authorization format" };
            return NextResponse.json({success:false, error: "Invalid authorization format", code:401}, {status: 401})
        }

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

        const user = await prisma.user.findUnique({
            where: {id:payload.id}
        })

        if (!user || !user.token) {
            return NextResponse.json({success:false, error: "User telah logout atau belum login", code:401}, {status: 401})
        }

        await prisma.user.update({
            where: { id: payload.id },
            data: {
                token: null
            },
        });

        return NextResponse.json({success:true, message: "logout successful", data: user}, {status: 200})


    } catch (error) {
        console.error(error)
        return NextResponse.json({success:false, error: "internal server erro", code:500}, {status: 500})
    }
}
```

Penjelasan :

**1. Inisialisasi dan Keamanan (Baris 1 - 5)**

- **jwtVerify**: Fungsi dari library jose yang digunakan untuk mendekripsi dan memvalidasi keaslian token JWT yang dikirim oleh pengguna.
- **const secret**: Mengonversi kode rahasia (JWT_SECRET) dari file environment menjadi format yang bisa dibaca oleh mesin (Uint8Array) untuk keperluan verifikasi token.

**2. Pengambilan Token dari Header (Baris 7 - 14)**

- **export async function PUT(request)**: Fungsi ini menggunakan metode **PUT**. Biasanya logout menggunakan POST atau DELETE, namun PUT di sini mungkin dipilih karena tujuannya adalah *mengupdate* status user di database.
- **authHeader**: Sistem mengambil data dari header HTTP bernama authorization.
- **Validasi Header**: Jika header kosong, sistem langsung menolak akses dengan status **401 (Unauthorized)**.

**3. Ekstraksi dan Validasi Format (Baris 16 - 20)**

- **authHeader.split(" ")**: Memisahkan teks "Bearer <token>".
- **Pengecekan Tipe**: Memastikan bahwa token yang dikirim adalah tipe **Bearer**. Jika formatnya salah (misal hanya mengirim token tanpa kata "Bearer"), sistem akan mengembalikan error.

**4. Verifikasi Keaslian Token (Baris 22 - 26)**

- **jwtVerify(token, secret)**: Langkah paling krusial. Sistem mengecek apakah token tersebut benar-benar dibuat oleh server Anda dan belum kedaluwarsa.
- **payload**: Jika verifikasi berhasil, sistem mengambil data yang tersimpan di dalam token (seperti ID user).

- **prisma.user.findUnique**: Mencari user di database berdasarkan ID yang didapat dari token tersebut.

**5. Validasi Status Login (Baris 28 - 30)**

- Sistem mengecek apakah user tersebut ada dan apakah di database kolom token-nya masih berisi. Jika kolom token di database sudah kosong (null), berarti user tersebut **sudah logout sebelumnya** atau memang belum login.

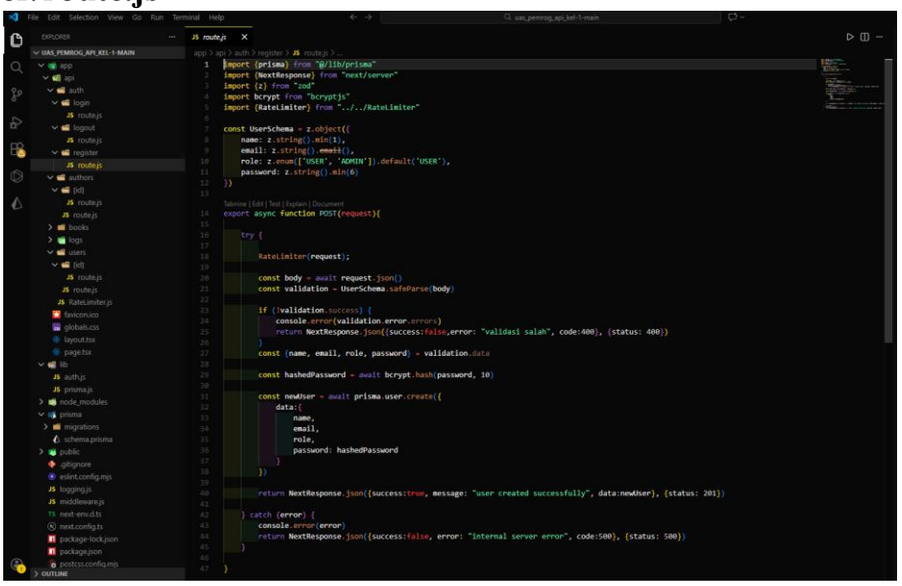
**6. Proses Logout - Menghapus Token (Baris 32 - 40)**

- **prisma.user.update**: Bagian inti dari logout. Sistem akan mencari user tersebut dan mengubah kolom **token menjadi null**.
- Dengan mengubah menjadi null, token yang dipegang pengguna secara teknis tidak lagi berlaku di sisi server.
- **Respon Berhasil**: Mengirimkan status **200 (OK)** dengan pesan "logout successful".

**7. Penanganan Error (Baris 42 - 46)**

- Jika terjadi kesalahan (misal token palsu atau database error), bagian catch akan menangkapnya dan mengirimkan respon error **500** atau membiarkan middleware menangani error autentikasi.

## C. register/route.js



```
import {prisma} from "@/lib/prisma"
import {NextResponse} from "next/server"
import {z} from "zod"
import bcrypt from "bcryptjs"
import {RateLimiter} from "../../RateLimiter"

const UserSchema = z.object({
    name: z.string().min(1),
    email: z.string().email(),
    role: z.enum(['USER', 'ADMIN']).default('USER'),
    password: z.string().min(6)
})

export async function POST(request){

    try {

        RateLimiter(request);

        const body = await request.json()
        const validation = UserSchema.safeParse(body)

        if (!validation.success) {
            console.error(validation.error.errors)
            return NextResponse.json({success:false,error: "validasi salah", code:400}, {status: 400})
        }
        const {name, email, role, password} = validation.data

        const hashedPassword = await bcrypt.hash(password, 10)

        const newUser = await prisma.user.create({
            data:{
                name,
                email,
                role,
                password: hashedPassword
            }
        })

        return NextResponse.json({success:true, message: "user created successfully", data:newUser}, {status: 201})

    } catch (error) {
        console.error(error)
        return NextResponse.json({success:false, error: "internal server error", code:500}, {status: 500})
    }
}
```

Penjelasan :

**1. Impor Modul dan Pustaka (Baris 1 - 5)**

- **prisma**: Digunakan untuk melakukan operasi *create* data ke database.
- **NextResponse**: Untuk mengirimkan respon HTTP ke client.
- **z (Zod)**: Pustaka untuk validasi skema data guna memastikan input dari user sesuai kriteria (misal: format email harus benar).
- **bcrypt**: Digunakan untuk melakukan *hashing* (enkripsi) password sebelum disimpan ke database agar aman.
- **RateLimiter**: Middleware untuk membatasi jumlah request guna mencegah penyalahgunaan API.

**2. Definisi Skema Validasi (Baris 7 - 12)**

Bagian ini mendefinisikan aturan main untuk data pendaftaran:

- **name**: Minimal 1 karakter.
- **email**: Harus dalam format email yang valid.
- **role**: Hanya boleh berisi 'USER' atau 'ADMIN', dengan nilai default adalah 'USER'.
- **password**: Minimal 6 karakter untuk keamanan.

**3. Ekstraksi dan Validasi Data (Baris 19 - 27)**

- **request.json()**: Mengambil data dari body request yang dikirim user.
- **UserSchema.safeParse(body)**: Melakukan validasi data terhadap skema Zod.
- **Pengecekan if (!validation.success)**: Jika data tidak valid (misal: password terlalu pendek), server akan mengirimkan error **400 (Bad Request)** dengan pesan "validasi salah".

**4. Enkripsi Password (Baris 29)**

- **bcrypt.hash(password, 10)**: Password asli dari user tidak boleh disimpan langsung. Baris ini mengubah password menjadi kode acak (hash). Angka 10 adalah *salt rounds* yang menentukan seberapa kuat enkripsinya.

**5. Penyimpanan ke Database (Baris 31 - 38)**

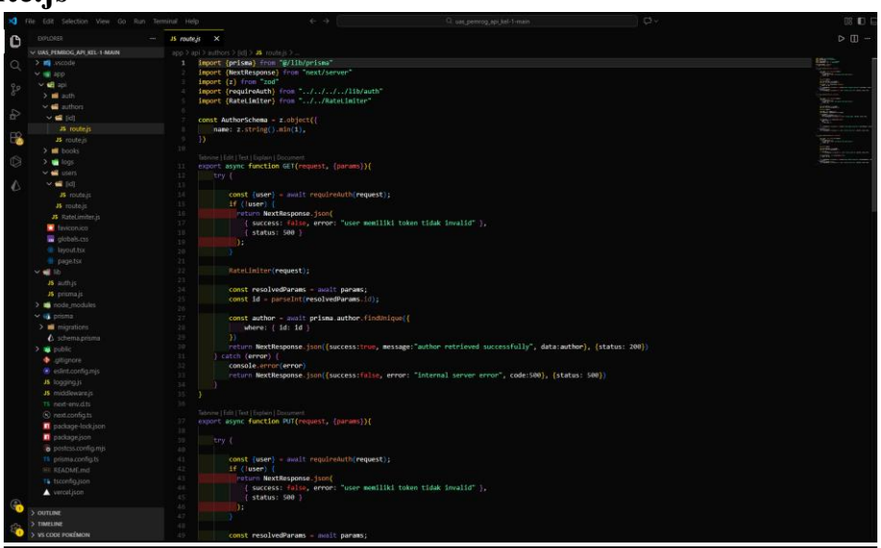
- **prisma.user.create**: Membuat record baru di tabel user.
- Sistem menyimpan name, email, role, dan hashedPassword yang sudah dienkripsi tadi.

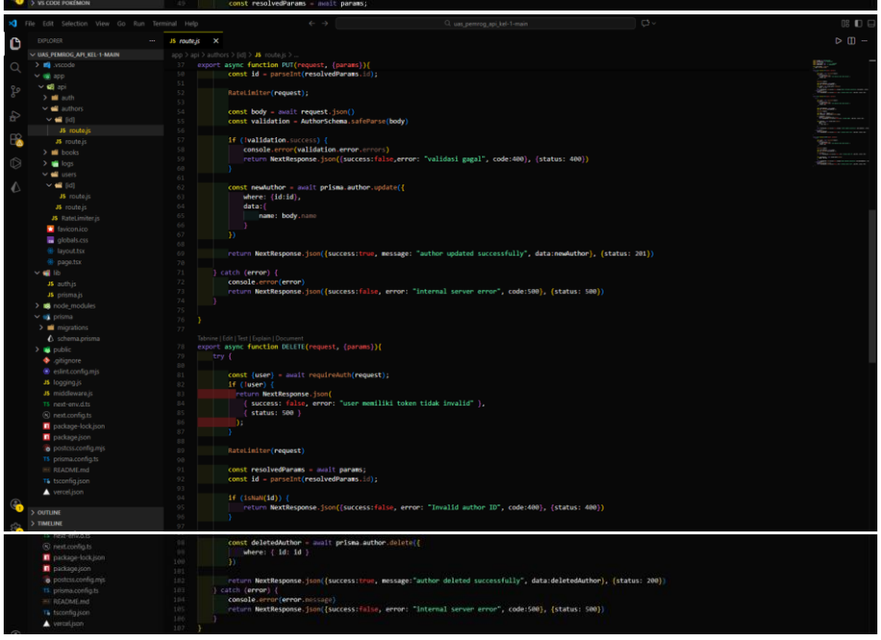
**6. Respon Berhasil dan Penanganan Error (Baris 40 - 46)**

- **Status 201**: Jika berhasil, sistem mengirimkan status **201 (Created)** yang menandakan data baru telah berhasil dibuat.
- **Error Handling (Catch)**: Jika terjadi kesalahan teknis (seperti email duplikat atau database terputus), sistem menangkapnya dan mengirimkan error **500 (Internal Server Error)**.

## 2. authors/[id]/…

### A. route.js





Penjelasan :

**1. Impor Modul dan Pustaka (Baris 1 - 5)**

- **prisma**: Digunakan untuk melakukan operasi *create* data ke database.
- **NextResponse**: Untuk mengirimkan respon HTTP ke client.
- **z (Zod)**: Pustaka untuk validasi skema data guna memastikan input dari user sesuai kriteria (misal: format email harus benar).
- **bcrypt**: Digunakan untuk melakukan *hashing* (enkripsi) password sebelum disimpan ke database agar aman.
- **RateLimiter**: Middleware untuk membatasi jumlah request guna mencegah penyalahgunaan API.

**2. Definisi Skema Validasi (Baris 7 - 12)**

Bagian ini mendefinisikan aturan main untuk data pendaftaran:

- **name**: Minimal 1 karakter.
- **email**: Harus dalam format email yang valid.

- **role**: Hanya boleh berisi 'USER' atau 'ADMIN', dengan nilai default adalah 'USER'.
- **password**: Minimal 6 karakter untuk keamanan.

**3. Ekstraksi dan Validasi Data (Baris 19 - 27)**

- **request.json()**: Mengambil data dari body request yang dikirim user.
- **UserSchema.safeParse(body)**: Melakukan validasi data terhadap skema Zod.
- **Pengecekan if (!validation.success)**: Jika data tidak valid (misal: password terlalu pendek), server akan mengirimkan error **400 (Bad Request)** dengan pesan "validasi salah".

**4. Enkripsi Password (Baris 29)**

- **bcrypt.hash(password, 10)**: Password asli dari user tidak boleh disimpan langsung. Baris ini mengubah password menjadi kode acak (hash). Angka 10 adalah *salt rounds* yang menentukan seberapa kuat enkripsinya.

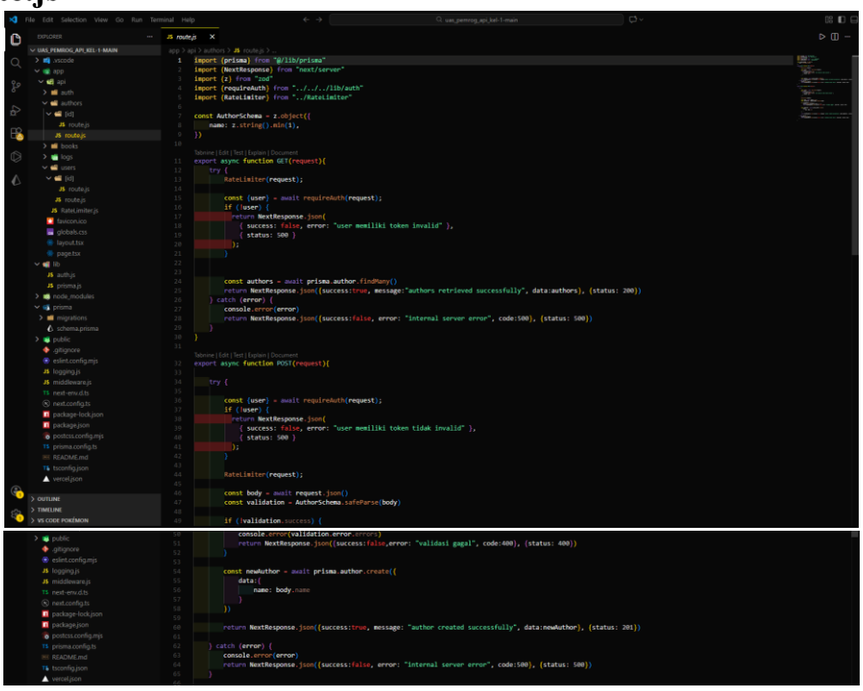
**5. Penyimpanan ke Database (Baris 31 - 38)**

- **prisma.user.create**: Membuat record baru di tabel user.
- Sistem menyimpan name, email, role, dan hashedPassword yang sudah dienkripsi tadi.

**6. Respon Berhasil dan Penanganan Error (Baris 40 - 46)**

- **Status 201**: Jika berhasil, sistem mengirimkan status **201 (Created)** yang menandakan data baru telah berhasil dibuat.
- **Error Handling (Catch)**: Jika terjadi kesalahan teknis (seperti email duplikat atau database terputus), sistem menangkapnya dan mengirimkan error **500 (Internal Server Error)**.

## B. route.js



Penjelasan :

Berikut adalah penjelasan untuk file **Authors Main Route** (app/api/authors/route.js). File ini menangani pengelolaan data penulis (author) secara kolektif, yaitu mengambil semua daftar penulis dan menambahkan penulis baru.

Sama seperti rute sebelumnya, file ini sangat mementingkan keamanan dengan mewajibkan login melalui requireAuth.

**1. Definisi Skema (Baris 7 - 9)**

- **AuthorSchema**: Menggunakan library **Zod** untuk memastikan bahwa saat menambah penulis baru, data name harus berupa teks (string) dan tidak boleh kosong (min(1)).

**2. Fungsi GET - Mengambil Semua Penulis (Baris 11 - 31)**

Fungsi ini digunakan untuk menampilkan daftar seluruh penulis yang ada di database.

- **RateLimiter(request)**: Membatasi kecepatan permintaan untuk mencegah beban berlebih pada server.
- **requireAuth(request)**: Memastikan hanya pengguna terautentikasi yang bisa menarik data ini. Jika token tidak valid, akan muncul error status **500** (walaupun secara logika seharusnya 401, kode Anda diatur ke 500).
- **prisma.author.findMany()**: Perintah database untuk mengambil **semua** baris data dari tabel author.
- **Respon**: Jika berhasil, mengirimkan data daftar penulis dengan status **200 (OK)**.

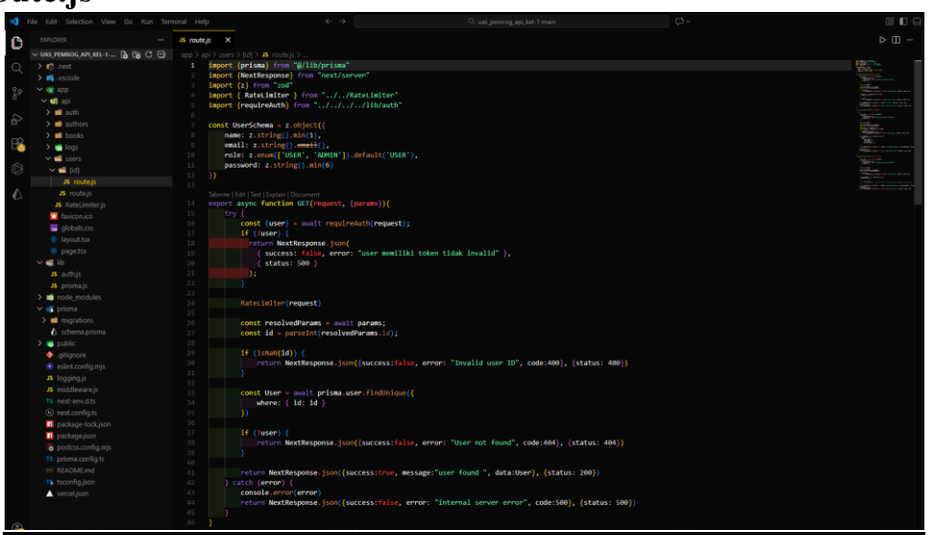
**3. Fungsi POST - Menambah Penulis Baru (Baris 32 - 66)**

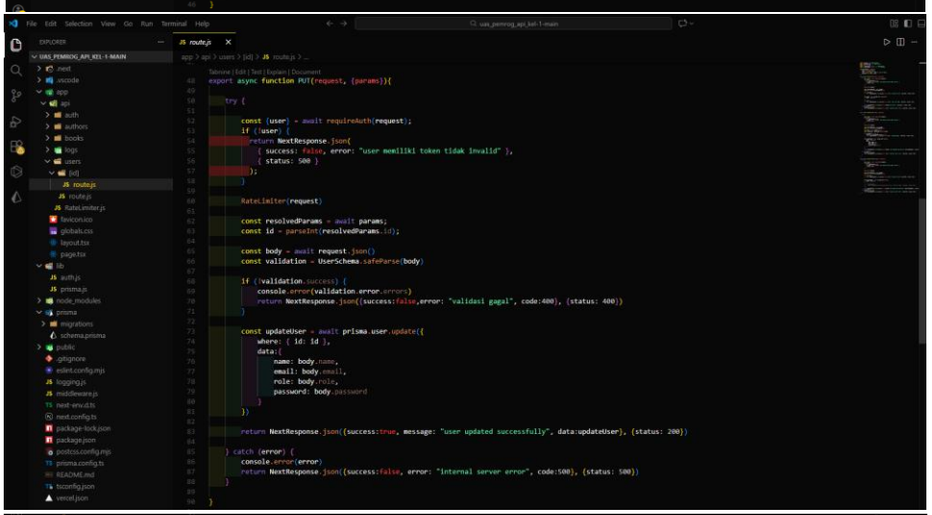
Fungsi ini digunakan untuk mendaftarkan nama penulis baru ke dalam sistem.

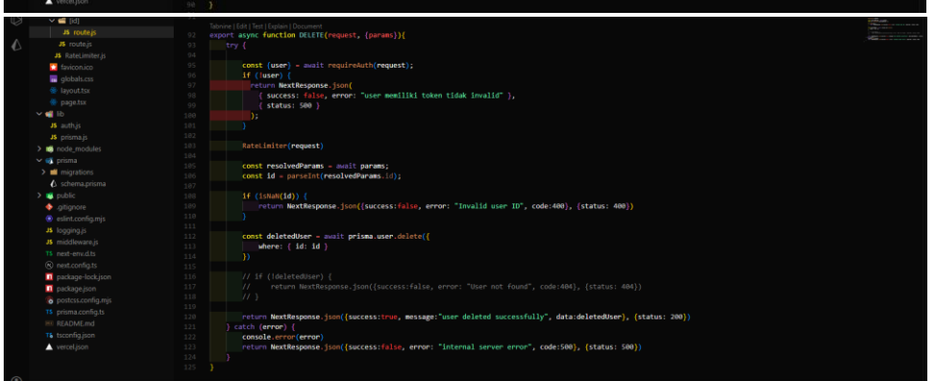
- **Proteksi Keamanan**: Kembali menggunakan requireAuth dan RateLimiter sebelum memproses data.
- **Validasi Input**:
    - request.json(): Mengambil data dari body permintaan.
    - AuthorSchema.safeParse(body): Mengecek apakah data sudah sesuai aturan (nama tidak kosong). Jika gagal, mengirim error **400 (Validasi Gagal)**.
- **prisma.author.create**: Memasukkan data nama baru ke kolom name di tabel author database.
- **Respon**: Mengirimkan data penulis yang baru dibuat dengan status **201 (Created)**.

## 3. users/[id]/…

### A. route.js







Penjelasan :

**1. Skema Validasi Pengguna**

Sebelum memproses data, sistem mendefinisikan aturan input menggunakan **Zod**:

- **name**: Wajib diisi, minimal 1 karakter.
- **email**: Harus berformat email yang valid.
- **role**: Terbatas pada pilihan 'USER' atau 'ADMIN', dengan default 'USER'.
- **password**: Minimal memiliki panjang 6 karakter.

**2. Fungsi GET - Mendapatkan Detail Pengguna**

Digunakan untuk mengambil informasi profil satu pengguna:

- **Keamanan**: Memeriksa autentikasi melalui requireAuth dan membatasi request dengan RateLimiter.
- **Proses**: Mengambil ID dari parameter URL, mengubahnya menjadi integer, dan melakukan pencarian di database menggunakan prisma.user.findUnique.
- **Respon**: Jika user tidak ditemukan, sistem mengembalikan status **404 (Not Found)**. Jika ditemukan, data user dikirim dengan status **200 (OK)**.

**3. Fungsi PUT - Memperbarui Data Pengguna**

Digunakan untuk mengubah informasi akun pengguna yang sudah ada:

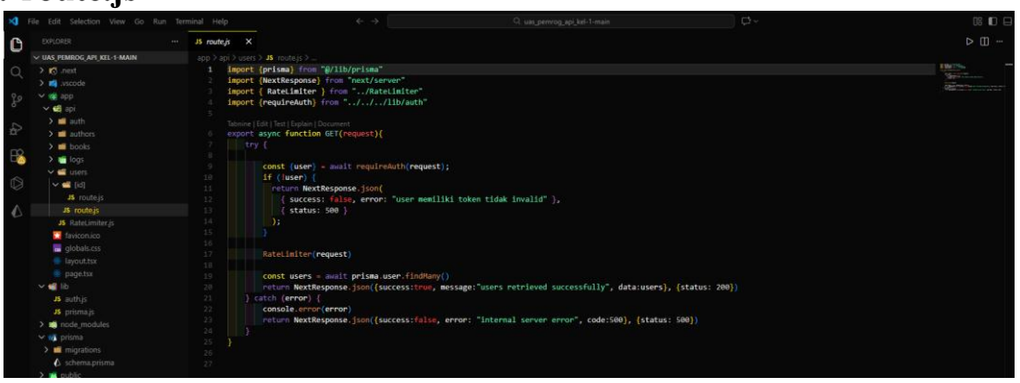
- **Validasi**: Data baru yang dikirim user diperiksa kesesuaiannya dengan UserSchema menggunakan safeParse.
- **Update Database**: Menggunakan prisma.user.update untuk memperbarui field nama, email, role, dan password berdasarkan ID pengguna tersebut.
- **Hasil**: Mengembalikan pesan sukses "user updated successfully" dengan status **200 (OK)**.

**4. Fungsi DELETE - Menghapus Pengguna**

Digunakan untuk menghapus akun dari sistem:

- **Pengecekan ID**: Sistem memastikan ID yang diterima adalah angka yang valid melalui fungsi isNaN(id). Jika bukan angka, akan muncul error **400 (Invalid user ID)**.
- **Eksekusi**: Menghapus baris data di tabel user menggunakan prisma.user.delete.
- **Konfirmasi**: Jika berhasil, mengembalikan status **200 (OK)** beserta data user yang telah dihapus.

## B. route.js




Penjelasan :

**1. Impor Modul & Proteksi Dasar (Baris 1 - 5)**

File ini menggunakan standar yang sama dengan modul lainnya untuk menjaga konsistensi:

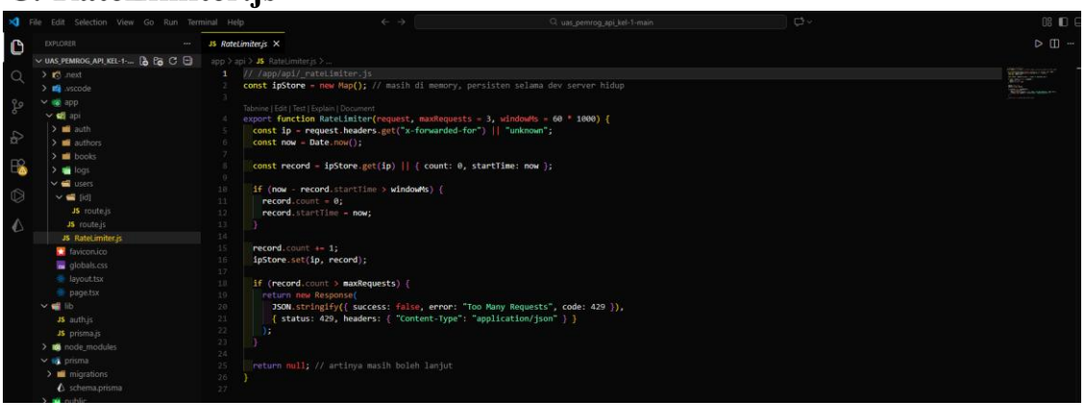
- **requireAuth**: Middleware wajib untuk memastikan bahwa data daftar pengguna hanya bisa diakses oleh orang yang sudah login.
- **RateLimiter**: Mencegah serangan *denial-of-service* atau *scraping* data user secara massal.

**2. Fungsi GET - Menampilkan Semua Pengguna (Baris 6 - 25)**

Fungsi ini digunakan oleh admin atau sistem untuk menarik seluruh database pengguna.

- **Proses Autentikasi**: Sistem menjalankan requireAuth(request) di awal. Jika gagal (user tidak ditemukan/token salah), sistem mengembalikan error status **500** dengan pesan "user memiliki token tidak invalid".
- **Pengambilan Data**: Menggunakan perintah **prisma.user.findMany()** untuk mengambil semua record dari tabel pengguna.
- **Respon Sukses**: Jika berhasil, data dikirimkan dalam format JSON dengan status **200 (OK)** dan pesan "users retrieved successfully".

## C. RateLimiter.js



```
// /app/api/_rateLimiter.js
const ipStore = new Map(); // masih di memory, persisten selama dev server hidup

export function RateLimiter(request, maxRequests = 3, windowMs = 60 * 1000) {
  const ip = request.headers.get("x-forwarded-for") || "unknown";
  const now = Date.now();

  const record = ipStore.get(ip) || { count: 0, startTime: now };

  if (now - record.startTime > windowMs) {
    record.count = 0;
    record.startTime = now;
  }

  record.count += 1;
  ipStore.set(ip, record);

  if (record.count > maxRequests) {
    return new Response(
      JSON.stringify({ success: false, error: "Too Many Requests", code: 429 }),
      { status: 429, headers: { "Content-Type": "application/json" } }
    );
  }

  return null; // artinya masih boleh lanjut
}
```

Penjelasan :

**1. Inisialisasi Penyimpanan (Baris 1 - 2)**

- **const ipStore = new Map()**: Sistem menggunakan objek Map untuk menyimpan data jumlah request pengguna secara sementara di memori server.
- **Catatan**: Karena disimpan di memori (in memory), data ini akan terhapus jika server dijalankan ulang (restart).

**2. Parameter Fungsi (Baris 4)**

Fungsi RateLimiter memiliki aturan default sebagai berikut:

- **maxRequests = 3**: Maksimal 3 kali percobaan.
- **windowMs = 60 * 1000**: Jangka waktu pembatasan adalah 60 detik (1 menit).

**3. Identifikasi Pengguna (Baris 5 - 8)**

- **request.headers.get("x-forwarded-for")**: Sistem mencoba mengambil alamat IP asli pengguna melalui header HTTP.
- **ipStore.get(ip)**: Sistem mengecek apakah IP tersebut sudah pernah melakukan request sebelumnya. Jika belum, data baru dibuat dengan hitungan (count) mulai dari 0.

**4. Logika Reset Waktu (Baris 10 - 13)**

- Sistem mengecek apakah waktu tunggu (1 menit) sudah berlalu.
- Jika sudah lewat dari satu menit sejak request terakhir, hitungan count akan diatur ulang kembali ke 0.

**5. Penambahan Hitungan & Validasi (Baris 15 - 23)**

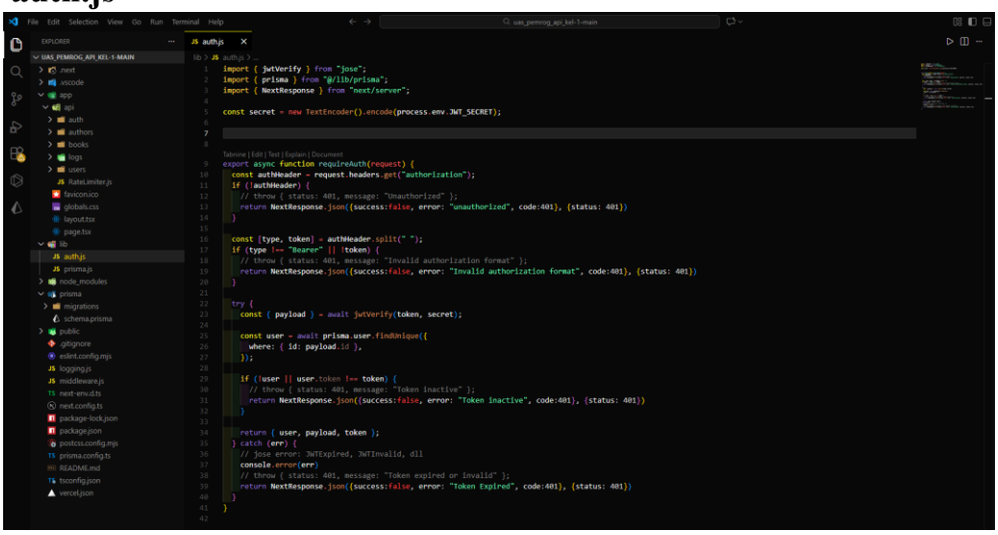
- Setiap kali pengguna mengakses API, nilai record.count akan bertambah 1.
- **Pengecekan Pelanggaran**: Jika count sudah melebihi 3, sistem akan langsung menolak permintaan.
- **Respon Error 429**: Jika ditolak, sistem mengirimkan status **429 (Too Many Requests)** beserta pesan error dalam format JSON.

**6. Izin Akses (Baris 25)**

- **return null**: Jika jumlah request masih di bawah batas (misal baru request ke-1 atau ke-2), fungsi akan mengembalikan nilai null, yang artinya proses API boleh dilanjutkan ke tahap berikutnya.

## 4. lib/…

### A. auth.js



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

const secret = new TextEncoder().encode(process.env.JWT_SECRET);

export async function requireAuth(request) {
  const authHeader = request.headers.get("authorization");
  if (!authHeader) {
    // throw { status: 401, message: "Unauthorized" };
    return NextResponse.json({success:false, error: "unauthorized", code:401}, {status: 401})
  }

  const [type, token] = authHeader.split(" ");
  if (type !== "Bearer" || !token) {
    // throw { status: 401, message: "Invalid authorization format" };
    return NextResponse.json({success:false, error: "Invalid authorization format", code:401}, {status: 401})
  }

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

    const user = await prisma.user.findUnique({
      where: { id: payload.id },
    });

    if (!user || user.token !== token) {
      // throw { status: 401, message: "Token inactive" };
      return NextResponse.json({success:false, error: "Token inactive", code:401}, {status: 401})
    }

    return { user, payload, token };
  } catch (err) {
    // jose error: JWTExpired, JWTInvalid, dll
    console.error(err);
    // throw { status: 401, message: "Token expired or invalid" };
    return NextResponse.json({success:false, error: "Token Expired", code:401}, {status: 401})
  }
}
```

Penjelasan :

**1. Inisialisasi dan Kunci Rahasia (Baris 1 - 5)**

- **jwtVerify**: Diimpor dari library jose untuk memeriksa apakah sebuah token JWT asli, belum dimanipulasi, dan belum kedaluwarsa.
- **secret**: Mengambil kunci rahasia dari environment variable (JWT_SECRET) dan mengubahnya menjadi format Uint8Array agar dapat digunakan oleh proses verifikasi digital.

**2. Pengambilan & Validasi Header (Baris 10 - 20)**

- **request.headers.get("authorization")**: Sistem mencari header bernama "authorization" yang dikirimkan oleh klien (browser/mobile app).
- **Cek Keberadaan Header**: Jika header tidak ada, fungsi langsung menghentikan proses dan mengembalikan status **401 (Unauthorized)** dengan pesan "unauthorized".
- **Format Bearer**: Sistem memisahkan string header (misal: "Bearer xyz123"). Jika kata pertama bukan "Bearer" atau tokennya kosong, sistem menolak akses karena formatnya tidak standar.

**3. Verifikasi Token & Pencarian User (Baris 22 - 28)**

- **jwtVerify(token, secret)**: Token didekripsi menggunakan kunci rahasia. Jika token palsu atau sudah lama, proses akan gagal dan lari ke blok catch.
- **payload**: Jika berhasil, data di dalam token (seperti ID user) diekstraksi.
- **prisma.user.findUnique**: Sistem mencocokkan ID dari token tersebut dengan data pengguna di database untuk memastikan akun tersebut benar-benar ada.

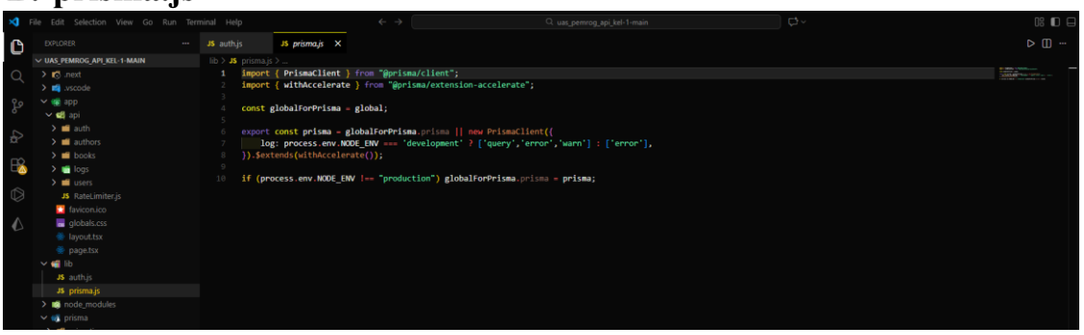
**4. Sinkronisasi Sesi Database (Baris 29 - 33)**

- **Cek Token Aktif**: Baris ini membandingkan token yang dikirim user dengan token yang tersimpan di database.
- Jika token di database sudah dihapus (null) atau berbeda (misal user sudah logout di perangkat lain), maka token dianggap **"Token inactive"** dan akses ditolak (status 401).

**5. Hasil Akhir & Penanganan Error (Baris 34 - 41)**

- **return { user, payload, token }**: Jika semua validasi lolos, fungsi mengembalikan objek berisi data user. Data inilah yang digunakan oleh file route.js lain (seperti Authors atau Users) untuk mengetahui siapa yang sedang mengakses.
- **catch (err)**: Jika token sudah kedaluwarsa atau rusak secara teknis, sistem menangkap error tersebut dan mengirimkan status **401** dengan pesan **"Token Expired"**.

## B. prisma.js




Penjelasan :

**. Inisialisasi dan Kunci Rahasia (Baris 1 - 5)**

- **jwtVerify**: Diimpor dari library jose untuk memeriksa apakah sebuah token JWT asli, belum dimanipulasi, dan belum kedaluwarsa.
- **secret**: Mengambil kunci rahasia dari environment variable (JWT_SECRET) dan mengubahnya menjadi format Uint8Array agar dapat digunakan oleh proses verifikasi digital.

**2. Pengambilan & Validasi Header (Baris 10 - 20)**

- **request.headers.get("authorization")**: Sistem mencari header bernama "authorization" yang dikirimkan oleh klien (browser/mobile app).
- **Cek Keberadaan Header**: Jika header tidak ada, fungsi langsung menghentikan proses dan mengembalikan status **401 (Unauthorized)** dengan pesan "unauthorized".
- **Format Bearer**: Sistem memisahkan string header (misal: "Bearer xyz123"). Jika kata pertama bukan "Bearer" atau tokennya kosong, sistem menolak akses karena formatnya tidak standar.

**3. Verifikasi Token & Pencarian User (Baris 22 - 28)**

- **jwtVerify(token, secret)**: Token didekripsi menggunakan kunci rahasia. Jika token palsu atau sudah lama, proses akan gagal dan lari ke blok catch.
- **payload**: Jika berhasil, data di dalam token (seperti ID user) diekstraksi.
- **prisma.user.findUnique**: Sistem mencocokkan ID dari token tersebut dengan data pengguna di database untuk memastikan akun tersebut benar-benar ada.

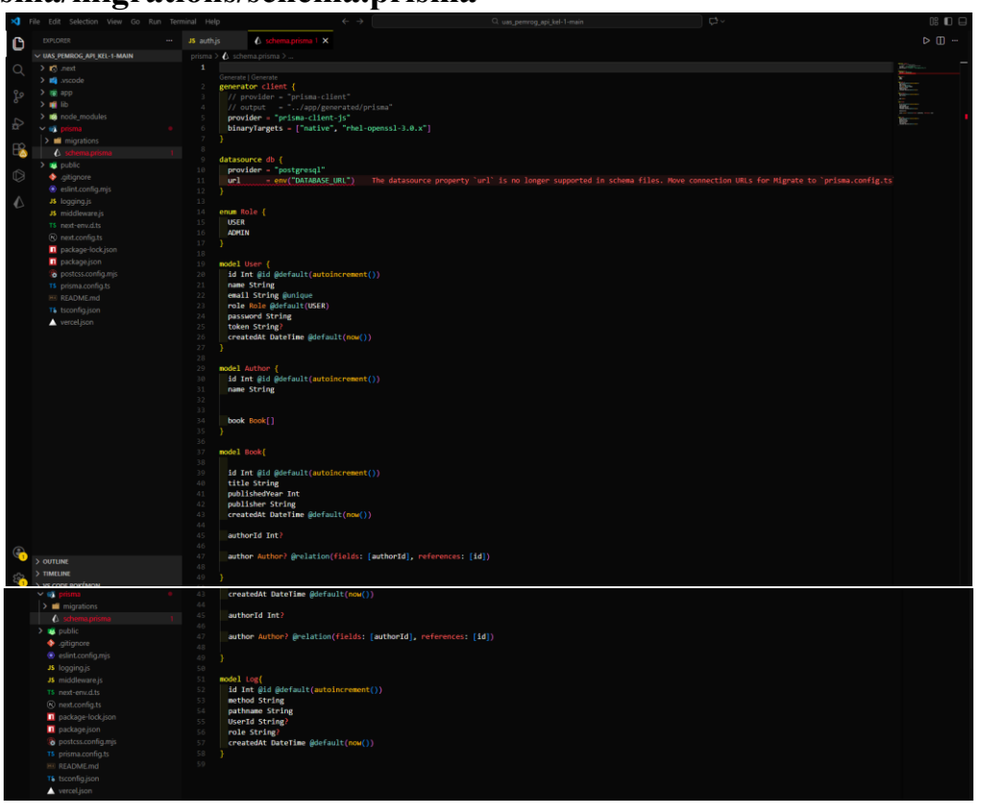
**4. Sinkronisasi Sesi Database (Baris 29 - 33)**

- **Cek Token Aktif**: Baris ini membandingkan token yang dikirim user dengan token yang tersimpan di database.
- Jika token di database sudah dihapus (null) atau berbeda (misal user sudah logout di perangkat lain), maka token dianggap **"Token inactive"** dan akses ditolak (status 401).

**5. Hasil Akhir & Penanganan Error (Baris 34 - 41)**

- **return { user, payload, token }**: Jika semua validasi lolos, fungsi mengembalikan objek berisi data user. Data inilah yang digunakan oleh file route.js lain (seperti Authors atau Users) untuk mengetahui siapa yang sedang mengakses.
- **catch (err)**: Jika token sudah kedaluwarsa atau rusak secara teknis, sistem menangkap error tersebut dan mengirimkan status **401** dengan pesan **"Token Expired"**.

## 5. prisma/migrations/schema.prisma



```
generator client {
  // provider = "prisma-client"
  // output   = "../app/generated/prisma"
  provider = "prisma-client-js"
  binaryTargets = ["native", "rhel-openssl-3.0.x"]
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")   The datasource property `url` is no longer supported in schema files. Move connection URLs for Migrate to `prisma.config.ts
}

enum Role {
  USER
  ADMIN
}

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

model Author {
  id Int @id @default(autoincrement())
  name String


  book Book[]
}

model Book{

  id Int @id @default(autoincrement())
  title String
  publishedYear Int
  publisher String
  createdAt DateTime @default(now())

  authorId Int?

  author Author? @relation(fields: [authorId], references: [id])

}
```



```
  createdAt DateTime @default(now())

  authorId Int?

  author Author? @relation(fields: [authorId], references: [id])

}
model Log{
  id Int @id @default(autoincrement())
  method String
  pathname String
  UserId String?
  role String?
  createdAt DateTime @default(now())
}
```

Penjelasan :

**1. Konfigurasi Generator & Datasource (Baris 1 - 13)**

- **generator client**: Menginstruksikan Prisma untuk membuat "Prisma Client", yaitu alat yang Anda gunakan di kode route.js (seperti prisma.user.findMany) agar bisa berinteraksi dengan database menggunakan bahasa Javascript/Typescript.
- **datasource db**: Menentukan bahwa database yang digunakan adalah **PostgreSQL**.
- **url = env("DATABASE_URL")**: Alamat koneksi database diambil dari file lingkungan (.env) agar informasi sensitif seperti password database tidak tersebar di kode program.

**2. Definisi Enum Role (Baris 14 - 17)**

- **enum Role**: Mendefinisikan pilihan tetap untuk peran pengguna, yaitu **USER** atau **ADMIN**. Ini memastikan tidak ada peran lain di luar dua pilihan tersebut yang bisa masuk ke database.

**3. Model User - Tabel Pengguna (Baris 19 - 27)**

Ini adalah tabel yang menyimpan data akun:

- **id**: Kunci utama (Primary Key) yang bertambah otomatis setiap ada user baru.
- **email**: Diberi label @unique, artinya tidak boleh ada dua pengguna dengan email yang sama.
- **role**: Menggunakan pilihan dari enum Role dengan nilai default adalah USER.
- **token**: Ditandai dengan tanda tanya (String?), artinya kolom ini bersifat **opsional** (bisa kosong/null), yang digunakan untuk menyimpan JWT saat login.

**4. Model Author & Book - Relasi Data (Baris 29 - 49)**

Bagian ini menjelaskan hubungan antara Penulis dan Buku:

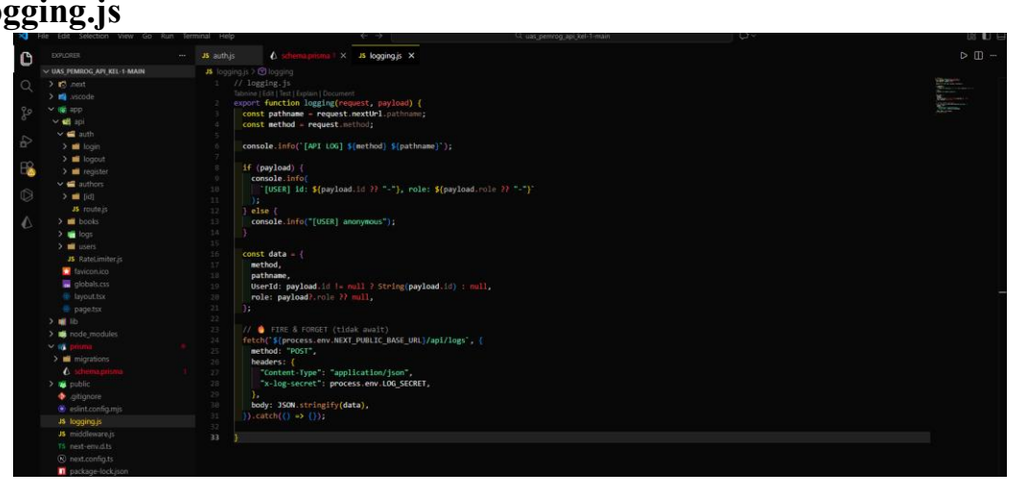
- **Model Author**: Menyimpan nama penulis dan memiliki daftar buku (book Book[]).
- **Model Book**: Menyimpan informasi detail buku seperti judul, tahun terbit, dan penerbit.
- **Relasi (One-to-Many)**: Seorang penulis bisa memiliki banyak buku. Hal ini dihubungkan melalui **authorId** di tabel Book yang merujuk pada **id** di tabel Author.

**5. Model Log - Pencatatan Aktivitas (Baris 51 - 58)**

Digunakan untuk keperluan audit atau pemantauan sistem:

- Menyimpan metode HTTP yang digunakan (method), alamat URL yang diakses (pathname), serta ID dan peran pengguna yang melakukan aksi tersebut.
- **createdAt**: Mencatat waktu otomatis saat log dibuat menggunakan fungsi now().

## 6. logging.js




Penjelasan :

**1. Definisi Fungsi dan Identifikasi Request (Baris 2 - 4)**

- **export function logging(request, payload)**: Fungsi ini menerima dua data utama: request (data lalu lintas HTTP) dan payload (data pengguna yang sudah login).
- **pathname & method**: Sistem secara otomatis mengambil alamat URL yang diakses (misal: /api/authors) dan metode yang digunakan (misal: POST atau GET).

**2. Output Konsol untuk Developer (Baris 6 - 14)**

- **console.info**: Menampilkan log aktivitas secara langsung di terminal server. Ini memudahkan developer memantau aplikasi secara *real-time*.
- **Identifikasi User**: Jika pengguna sudah login, log akan menampilkan **ID** dan **Role** mereka. Jika belum login (misal saat mengakses halaman login), sistem mencatatnya sebagai **"USER anonymous"**.

**3. Persiapan Data Log (Baris 16 - 21)**

Sistem menyusun objek data yang akan disimpan secara permanen ke database:

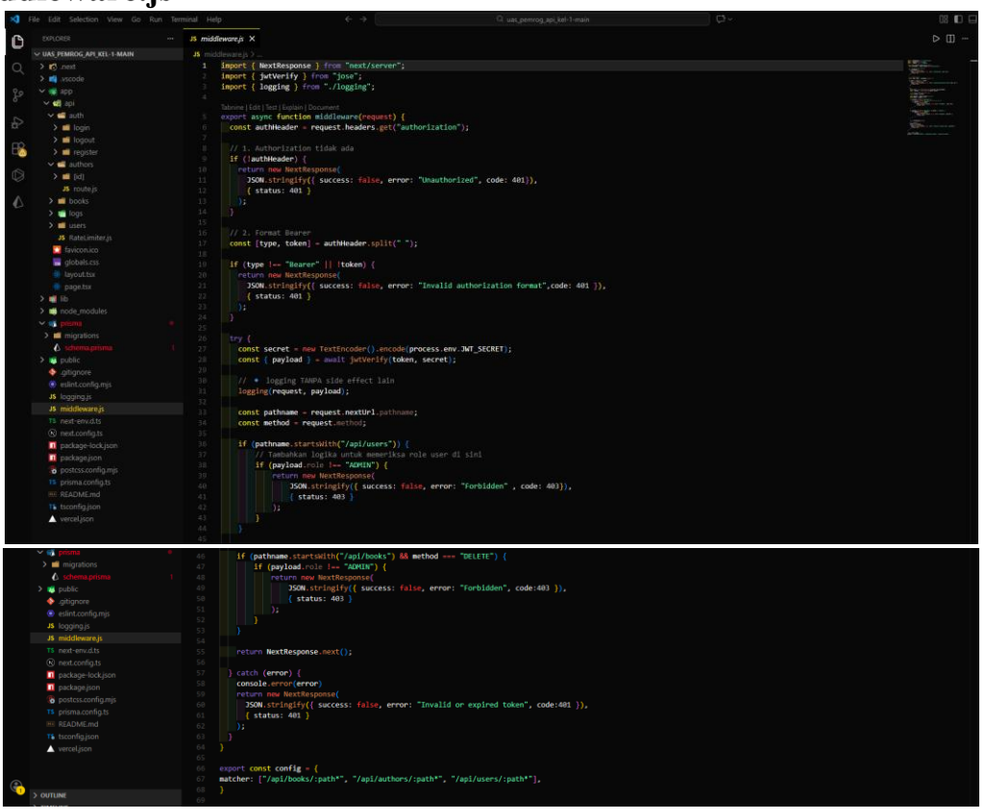
- **method & pathname**: Jenis aksi dan lokasi akses.
- **UserId & role**: Identitas pelaku aksi. Jika tidak ada (anonim), data ini akan diisi dengan null.

**4. Pengiriman Data ke API Logs (Baris 24 - 32)**

Bagian ini menggunakan metode **"Fire & Forget"**:

- **fetch(...)**: Sistem mengirimkan data log ke endpoint internal /api/logs.
- **x-log-secret**: Untuk keamanan, pengiriman log ini dilindungi oleh kunci rahasia (LOG_SECRET) agar tidak ada orang luar yang bisa memalsukan data log.
- **Tanpa await**: Proses ini dilakukan di latar belakang tanpa menunggu selesai, sehingga tidak menghambat kecepatan respon aplikasi bagi pengguna.

## 7. middleware.js



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

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

  // 1. Authorization tidak ada
  if (!authHeader) {
    return new NextResponse(
      JSON.stringify({ success: false, error: "Unauthorized", code: 401}),
      { status: 401 }
    );
  }

  // 2. Format Bearer
  const [type, token] = authHeader.split(" ");

  if (type !== "Bearer" || !token) {
    return new NextResponse(
      JSON.stringify({ success: false, error: "Invalid authorization format",code: 401 }),
      { status: 401 }
    );
  }

  try {
    const secret = new TextEncoder().encode(process.env.JWT_SECRET);
    const { payload } = await jwtVerify(token, secret);

    // ✅ logging TANPA side effect lain
    logging(request, payload);

    const pathname = request.nextUrl.pathname;
    const method = request.method;

    if (pathname.startsWith("/api/users")) {
      // Tambahkan logika untuk memeriksa role user di sini
      if (payload.role !== "ADMIN") {
        return new NextResponse(
          JSON.stringify({ success: false, error: "Forbidden" , code: 403}),
          { status: 403 }
        );
      }
    }

    if (pathname.startsWith("/api/books") && method === "DELETE") {
      if (payload.role !== "ADMIN") {
        return new NextResponse(
          JSON.stringify({ success: false, error: "Forbidden", code:403 }),
          { status: 403 }
        );
      }
    }

    return NextResponse.next();

  } catch (error) {
    console.error(error)
    return new NextResponse(
      JSON.stringify({ success: false, error: "Invalid or expired token", code:401 }),
      { status: 401 }
    );
  }
}

export const config = {
  matcher: ["/api/books/:path*", "/api/authors/:path*", "/api/users/:path*"],
}
```

Penjelasan :

**1. Verifikasi Token Global (Baris 5 - 24)**

Middleware ini memeriksa kredensial pengguna di setiap akses ke rute yang dilindungi:

- **Cek Header**: Mencari header authorization. Jika tidak ada, akses ditolak dengan status **401 (Unauthorized)**.
- **Format Bearer**: Memastikan token menggunakan format standar "Bearer [token]".
- **Dekripsi JWT**: Menggunakan kunci rahasia (JWT_SECRET) untuk memvalidasi isi token (payload).

**2. Integrasi Logging (Baris 30 - 31)**

- **logging(request, payload)**: Setelah token berhasil diverifikasi, middleware langsung memanggil fungsi logging untuk mencatat siapa yang mengakses, apa metodenya, dan ke URL mana mereka pergi. Ini memastikan setiap aktivitas terekam secara otomatis tanpa perlu menulis kode log di setiap file API.

**3. Otorisasi Berbasis Peran / Role-Based Access Control (Baris 36 - 52)**

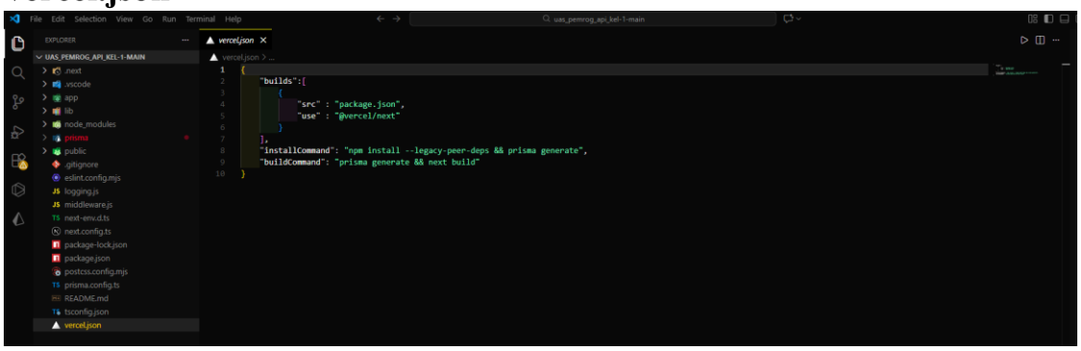
Middleware ini melakukan pembatasan akses berdasarkan level pengguna (Admin vs User):

- **Proteksi Data User**: Jika ada yang mencoba mengakses /api/users, middleware mengecek apakah payload.role adalah **ADMIN**. Jika bukan admin, akses ditolak dengan status **403 (Forbidden)**.
- **Proteksi Penghapusan Buku**: Jika ada permintaan untuk menghapus buku (DELETE pada /api/books), hanya pengguna dengan peran **ADMIN** yang diizinkan lewat.

**4. Konfigurasi Matcher (Baris 66 - 69)**

- **matcher**: Menentukan rute mana saja yang harus melewati "satpam" middleware ini.
- Berdasarkan kode Anda, middleware ini aktif untuk semua rute di bawah:
    - /api/books/:path*
    - /api/authors/:path*
    - /api/users/:path*.

## 8. vercel.json




Penjelasan :

**1. Proses Autentikasi Login (POST)**

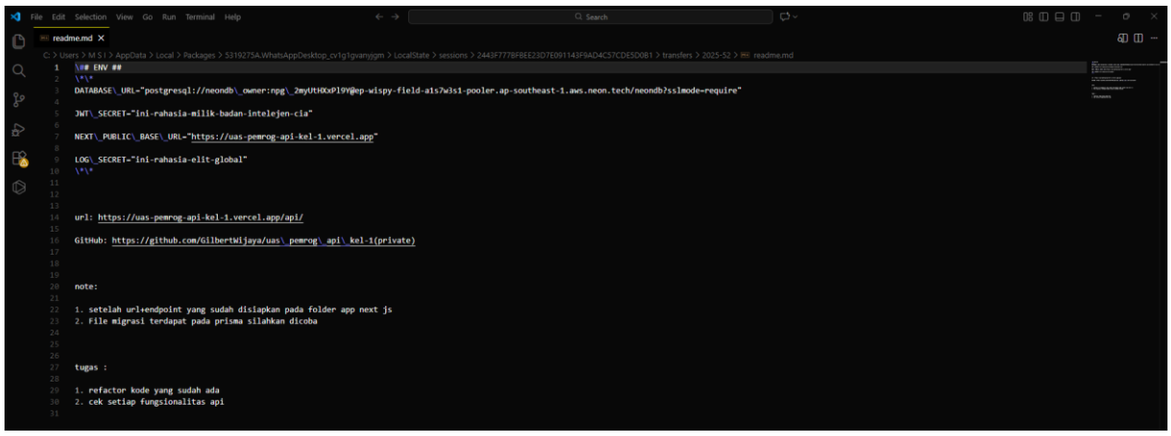
Fungsi ini menangani permintaan masuk dengan langkah-langkah keamanan yang ketat:

- **Rate Limiting**: Hal pertama yang dilakukan adalah menjalankan RateLimiter(request) untuk mencegah serangan *brute-force* yang mencoba menebak password berkali-kali dalam waktu singkat.
- **Pencarian Pengguna**: Sistem mengambil email dari body permintaan dan mencarinya di database menggunakan prisma.user.findUnique. Jika email tidak terdaftar, sistem langsung mengembalikan error **401 (Unauthorized)** dengan pesan yang ambigu ("Invalid email or password") demi alasan keamanan agar penyerang tidak tahu apakah email tersebut valid atau tidak.
- **Verifikasi Password**: Menggunakan bcrypt.compare, sistem mencocokkan password yang diinput dengan password terenkripsi (hash) yang ada di database.
- **Pembuatan JWT (JSON Web Token)**: Jika password cocok, sistem membuat token baru menggunakan library jose. Token ini berisi informasi identitas pengguna (id, email, dan role) dan diatur akan kedaluwarsa dalam **30 menit** (30m).

**2. Sinkronisasi Sesi Database**

- **Penyimpanan Token**: Setelah token JWT dibuat, sistem memperbarui kolom token pada tabel pengguna di database. Langkah ini sangat penting karena fungsi keamanan lainnya (seperti requireAuth) akan mencocokkan token yang dibawa pengguna dengan token yang tersimpan di sini untuk memastikan sesi tersebut masih aktif.

## 9. README.md




Penjelasan :

**Bagian 1: Variabel Lingkungan (.env)**

Bagian ini menyimpan kunci rahasia yang tidak boleh dipublikasikan ke GitHub agar aplikasi tetap aman.

- **Baris 1 (DATABASE_URL)**: Ini adalah *connection string* untuk menghubungkan aplikasi dengan database PostgreSQL yang dihosting di **Neon.tech** (AWS region ap-southeast-1).
    - Mengandung informasi: Protokol (postgresql), username (neondb_owner), password, host server, dan nama database (neondb).
- **Baris 3 (JWT_SECRET)**: Kunci rahasia yang digunakan oleh library jose atau jsonwebtoken untuk menandatangani (signing) token login pengguna.
    - Jika kunci ini bocor, orang lain bisa memalsukan identitas admin/user di aplikasi Anda.
- **Baris 5 (NEXT_PUBLIC_BASE_URL)**: Alamat dasar (base URL) aplikasi saat sudah dideploy ke **Vercel**.
    - Awalan NEXT_PUBLIC_ menandakan variabel ini bisa diakses oleh sisi Client (browser), bukan hanya server.
- **Baris 7 (LOG_SECRET)**: Kunci keamanan khusus yang digunakan oleh sistem logging.js dan API logs.
    - Ini memastikan hanya aplikasi internal Anda yang bisa mengirimkan data aktivitas ke database log.

**Bagian 2: Akses & Repositori**

- **Baris 10-11 (URL API)**: Endpoint utama untuk berinteraksi dengan sistem adalah https://uas-pemrog-api-kel-1.vercel.app/api/.
- **Baris 13-14 (GitHub)**: Alamat repositori kode sumber. Status "private" menunjukkan bahwa kode ini dilindungi dan hanya bisa diakses oleh tim pengembang.

**Bagian 3: Instruksi Operasional (Note)**

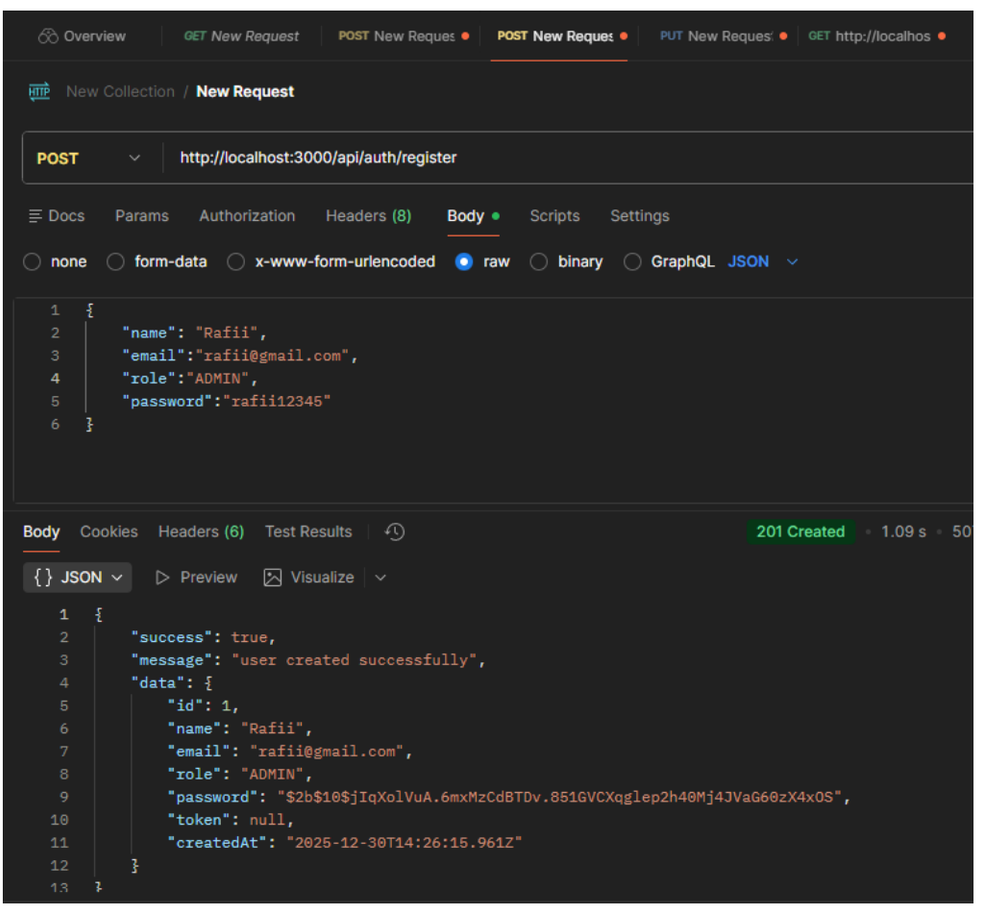
- **Baris 17-18 (Endpoint)**: Menjelaskan bahwa struktur API mengikuti struktur folder di dalam folder app Next.js (App Router).
- **Baris 19-20 (Migrasi)**: Memberitahu pengembang bahwa definisi tabel sudah ada di folder prisma.
    - Untuk menjalankannya, Anda perlu menggunakan perintah npx prisma migrate dev agar tabel User, Author, Book, dan Log terbentuk di Neon PostgreSQL.

**Bagian 4: Rencana Tugas (Task)**

- **Baris 22-24 (Refactor)**: Perintah untuk membersihkan kode, memperbaiki *typo* (seperti pesan error "token tidak invalid"), dan mengoptimalkan logika agar lebih efisien.
- **Baris 25-26 (Fungsionalitas)**: Perintah untuk melakukan testing pada setiap metode (GET, POST, PUT, DELETE) di semua endpoint (auth, authors, books, users, logs).

## 2.2 HASIL TESTING

### 1. Register




Penjelasan:

Dari gambar diatas menunjukkan hasil pengujian fungsionalitas Registrasi Pengguna Baru menggunakan REST Client. Pengujian ini bertujuan untuk memverifikasi apakah sistem mampu menerima data pengguna baru, menyimpannya ke dalam database PostgreSQL, serta menerapkan enkripsi pada password sesuai spesifikasi keamanan yang dirancang.

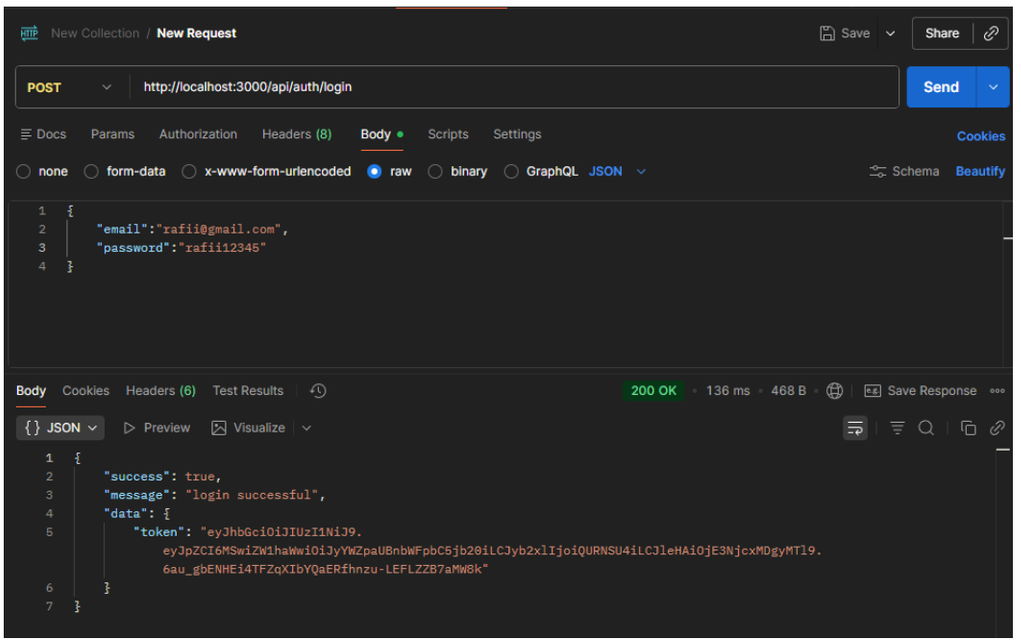
Detail Pengujian:

- Endpoint: http://localhost:3000/api/auth/register
- Method: POST
- Payload (Body): Data dikirim dalam format JSON yang berisi name, email, role, dan password (plaintext).

Analisis Hasil:

- Status Code 201 (Created): Server mengembalikan status HTTP 201, yang menandakan bahwa permintaan berhasil diproses dan resource baru (data user) telah sukses dibuat di dalam database.
- Keamanan Password (Bcrypt): Pada bagian Request Body, password yang dikirimkan adalah teks biasa ("rafii12345"). Namun, pada Response Body, field password yang dikembalikan telah berubah menjadi string acak yang diawali dengan $2b$10$.... Hal ini membuktikan bahwa implementasi password hashing menggunakan bcrypt telah berfungsi dengan baik. Password asli tidak disimpan secara mentah di database, menjaga keamanan akun pengguna.
- Validasi Role-Based Authorization (RBAC): Sistem berhasil memproses input role: "ADMIN". Dalam arsitektur aplikasi ini, penetapan peran sangat krusial karena menentukan hak akses terhadap endpoint selanjutnya.
    - Peran ADMIN: Akun yang didaftarkan pada pengujian di atas akan memiliki hak istimewa (privileged access) untuk melakukan mutasi data, yaitu mengakses endpoint POST untuk menambah data Buku (Books) dan data Penulis (Authors).
    - Peran USER: Sebaliknya, jika registrasi dilakukan dengan role: "USER", akun tersebut akan dibatasi hanya pada metode GET (Read-only), yang artinya hanya diperbolehkan untuk melihat daftar buku dan detail penulis tanpa izin untuk mengubah data.
- Response Data: Respons JSON mengembalikan objek user yang baru dibuat beserta id dan createdAt, serta pesan konfirmasi "user created successfully", yang memudahkan sisi frontend untuk memvalidasi keberhasilan proses.

## 2. Login




Penjelasan:

Gambar di atas memperlihatkan proses pengujian autentikasi pengguna menggunakan REST Client. Pengujian ini bertujuan untuk memverifikasi apakah sistem mampu memvalidasi kredensial pengguna (email dan password) yang telah terdaftar serta menerbitkan JSON Web Token (JWT) sebagai tanda otorisasi yang sah.

Detail Pengujian:

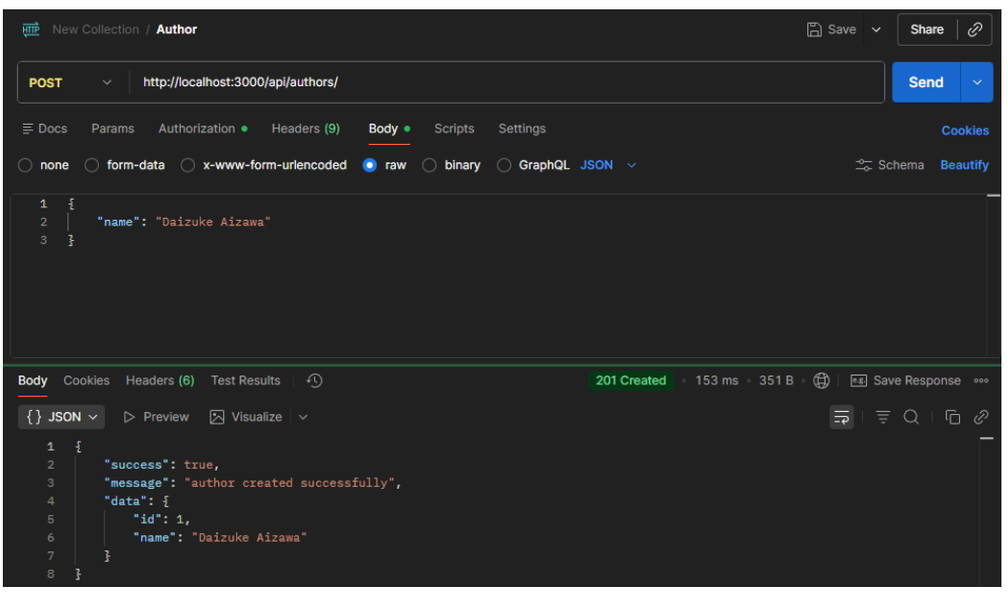
- Endpoint: http://localhost:3000/api/auth/login
- Method: POST
- Payload (Body): Mengirimkan kredensial akun yang baru dibuat, yaitu email ("rafii@gmail.com") dan password ("rafii12345").

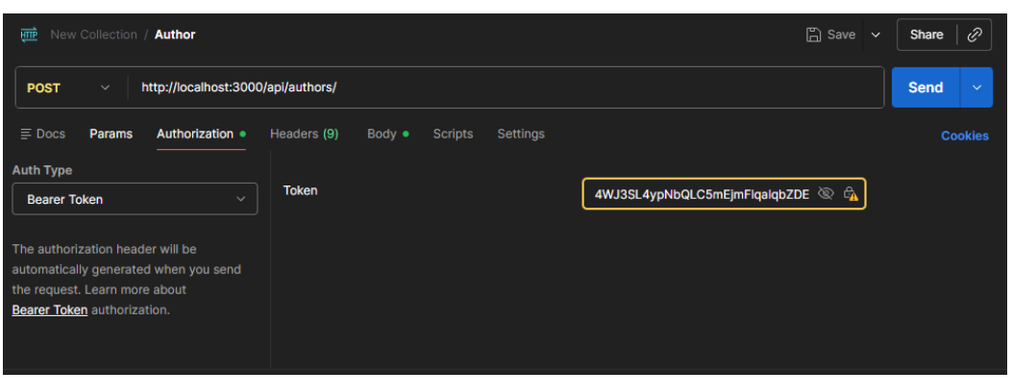
Analisis Hasil:

- Verifikasi Kredensial (Status 200 OK): Server mengembalikan status HTTP 200, yang menandakan bahwa proses login berhasil. Secara internal, sistem telah melakukan pencocokan antara email yang dikirim dengan data di database, serta memverifikasi kecocokan password (plaintext) dengan hash password yang tersimpan menggunakan algoritma bcrypt.
- Penerbitan JWT (Bearer Token): Pada bagian data di respons JSON, terdapat properti token yang berisi string panjang diawali dengan *eyJ...*. Ini adalah JWT (JSON Web Token) yang telah di-sign oleh server. Token ini membungkus informasi penting (payload) seperti User ID dan Role (dalam kasus ini: ADMIN) secara terenkripsi.

- Mekanisme Stateless Authentication: Keberadaan token ini krusial karena aplikasi menggunakan arsitektur stateless. Server tidak menyimpan sesi login di memori. Sebagai gantinya, token ini akan digunakan oleh sisi client (Frontend/Postman) sebagai "kunci akses". Untuk setiap permintaan berikutnya (seperti Menambah Buku atau Mengedit Author), token ini wajib disertakan di dalam Header Authorization dengan format Bearer <token>.
- Keamanan Hak Akses: Karena akun yang login adalah akun dengan role ADMIN (berdasarkan pengujian registrasi sebelumnya), token yang dihasilkan ini secara implisit membawa izin administrator. Sistem Middleware nantinya akan mendekode token ini untuk mengizinkan akses ke rute-rute yang diproteksi khusus Admin.

### 3. Tambah Data Author




Penjelasan:

Dari gambar diatas, sebelum dapat menambahkan data buku, sistem mengharuskan adanya data Penulis (*Author*) terlebih dahulu. Hal ini dikarenakan skema basis data menggunakan prinsip *Relational Integrity*, di mana setiap buku

harus terhubung dengan ID penulis yang valid. Pengujian ini dilakukan oleh pengguna dengan peran ADMIN.

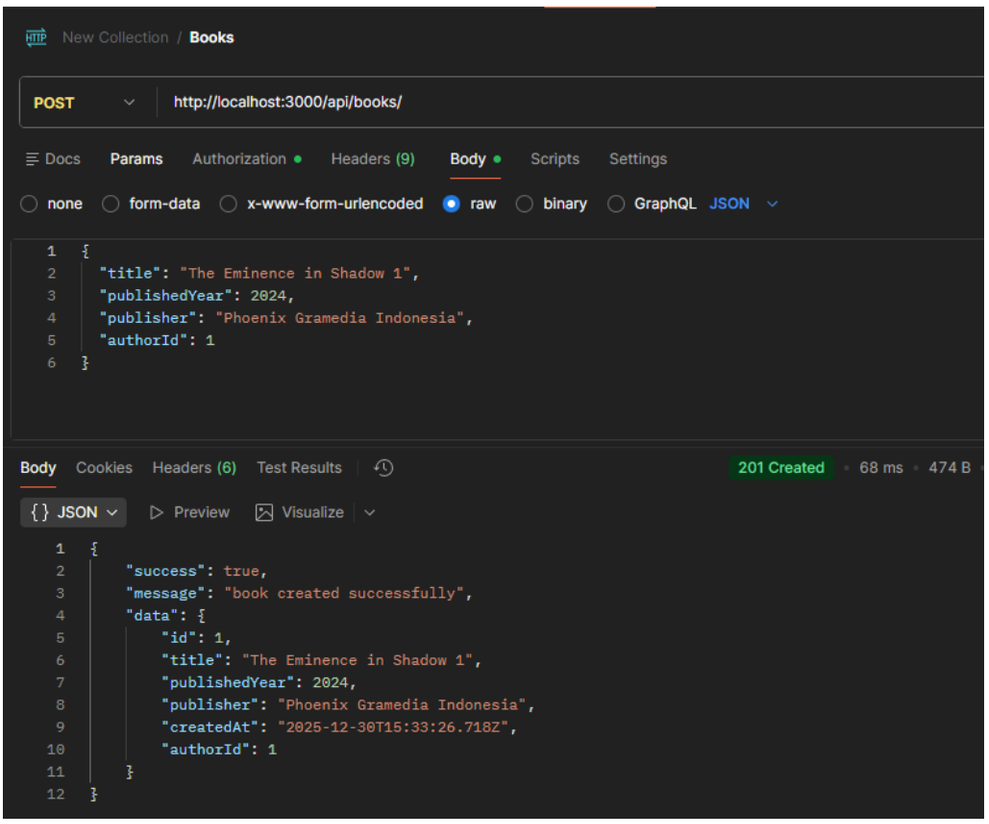
Detail Pengujian:

- Endpoint: http://localhost:3000/api/authors/
- Method: POST
- Authorization: Bearer Token (Token Admin dari login sebelumnya).

Analisis Hasil:

- Validasi Keamanan dan Middleware (Authorization): Seperti yang terlihat pada tangkapan layar (Tab Authorization), permintaan ini menyertakan Bearer Token pada Header Authorization. Token ini adalah JWT yang didapatkan saat proses login.
    - Fungsi Middleware: Sebelum data diproses, middleware di sisi server memverifikasi validitas token tersebut. Selain itu, middleware juga memastikan bahwa token tersebut milik pengguna dengan role ADMIN. Jika pengguna biasa (User) mencoba mengakses endpoint ini, akses akan ditolak (Forbidden), menjaga integritas data referensi perpustakaan.
- Pemrosesan Data Referensi: Pada bagian Body, dikirimkan data JSON sederhana berisi { "name": "Daizuke Aizawa" }. Server menerima input ini dan menggunakan Prisma ORM untuk membuat entitas baru di tabel Author.
- Respons Sistem dan Auto-Increment ID: Server mengembalikan status HTTP 201 Created beserta objek data penulis yang baru dibuat.
    - Catatan: Respons mengembalikan "id": 1. ID ini dihasilkan secara otomatis (*Auto-increment*) oleh database PostgreSQL. Nilai **ID inilah yang nantinya wajib digunakan** sebagai *Foreign Key* saat admin ingin menambahkan buku yang ditulis oleh penulis tersebut.

### 4. Tambah Data Buku




Penjelasan:

Setelah data referensi penulis tersedia, pengujian dilanjutkan dengan menambahkan data buku ke dalam sistem. Pengujian ini memvalidasi kesesuaian tipe data (*Type Safety*) dan integritas relasi (*Referential Integrity*) antar tabel dalam database.

Detail Pengujian:

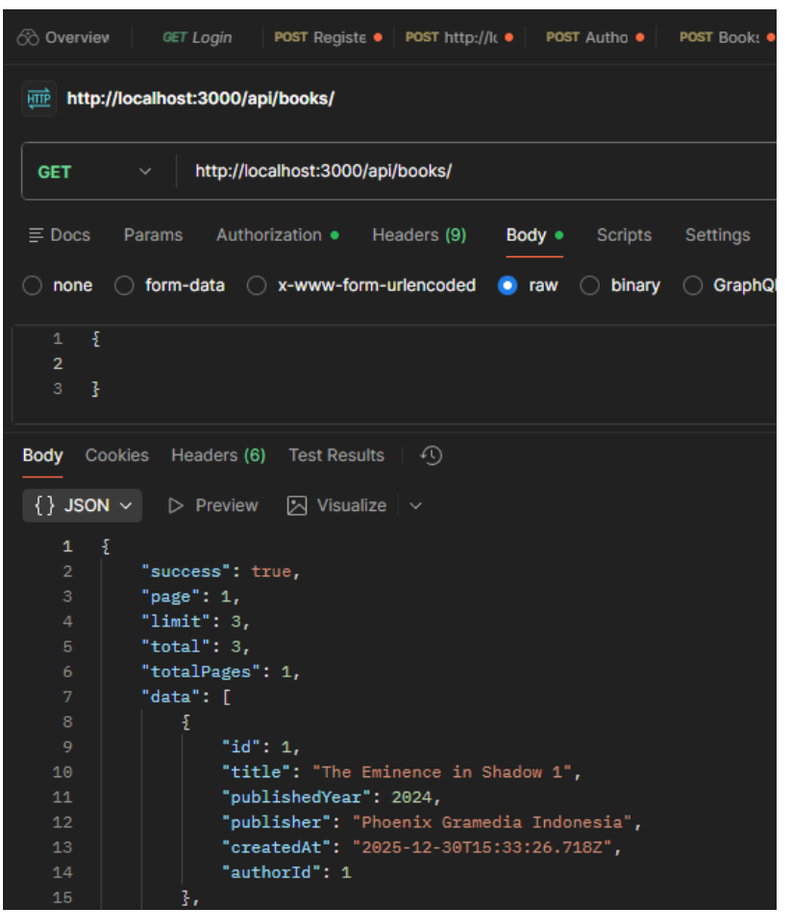
- Endpoint: http://localhost:3000/api/books/
- Method: POST
- Payload (Body): Mengirimkan data buku dengan format JSON yang telah disesuaikan dengan skema Prisma, khususnya pada tipe data integer.

Analisis Hasil

- Validasi Tipe Data (Schema Validation): Berbeda dengan percobaan awal yang mengalami kegagalan validasi, pada pengujian ini data publishedYear dikirim sebagai Integer (2024) dan bukan String. Hal ini sesuai dengan definisi model di schema.prisma. Keberhasilan server memproses data ini menunjukkan bahwa sistem validasi input di sisi backend berfungsi efektif dalam mencegah masuknya data yang tidak sesuai format.

- Implementasi Relasi (Foreign Key Association): Field authorId: 1 yang dikirimkan dalam request body berfungsi sebagai Foreign Key. Sistem berhasil menghubungkan buku "The Eminence in Shadow 1" dengan penulis "Daisuke Aizawa" (yang memiliki ID 1). Jika ID penulis tidak ditemukan di database, Prisma ORM akan menolak permintaan ini. Kesuksesan penyimpanan data membuktikan bahwa relasi One-to-Many antara entitas Author dan Book telah terjalin dengan benar.
- Persistensi Data: Server mengembalikan respons dengan status HTTP 201 Created dan menampilkan objek buku yang baru disimpan lengkap dengan id dan createdAt. Ini mengonfirmasi bahwa data telah tersimpan secara permanen di database PostgreSQL.

**5. Pengujian Endpoint Melihat Daftar Buku (GET)**




Penjelasan:

Setelah data buku tersedia di database, dilakukan pengujian fitur Read (Melihat Data). Sesuai dengan skenario hak akses, pengujian ini dilakukan menggunakan akun dengan peran USER. Tujuannya adalah memastikan bahwa pengguna umum dapat mengakses katalog buku namun tetap dalam batasan hanya membaca (Read-only). Namun, seharusnya sama aja fungsinya untuk peran ADMIN dibagian GET ini.

Detail Pengujian:

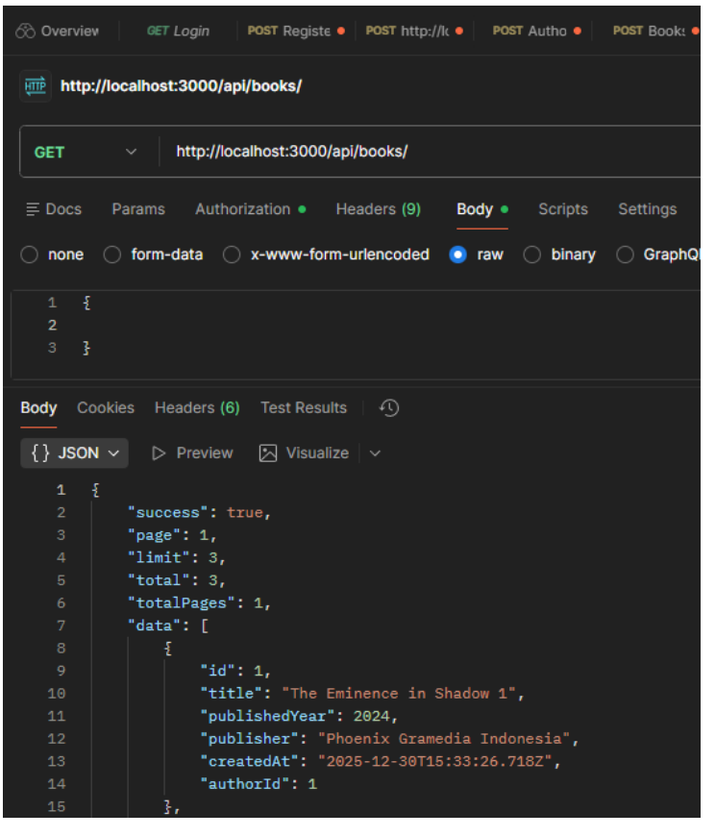
- Endpoint: http://localhost:3000/api/books/
- Method: GET
- Authorization: Bearer Token (Menggunakan Token akun User Biasa).

Analisis Hasil:

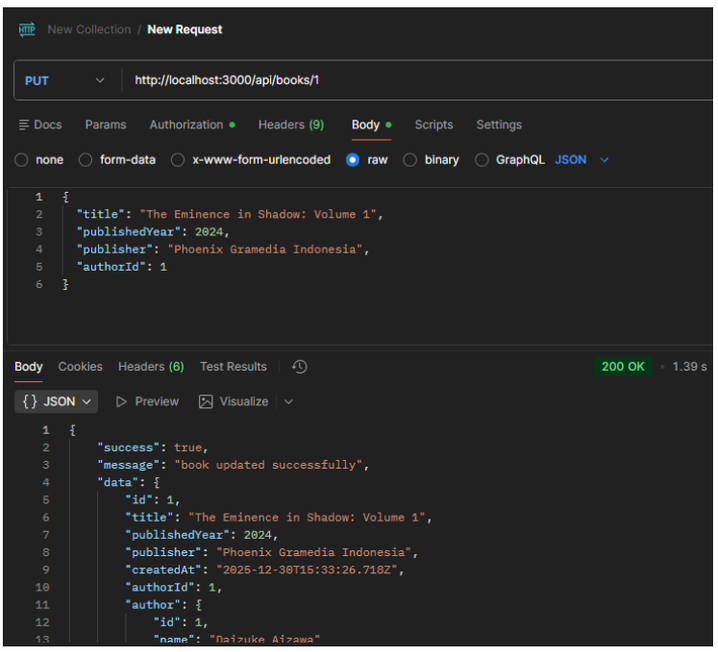
- Akses Data Sukses (Read Permission): Server memberikan respons dengan status sukses ("success": true). Hal ini membuktikan bahwa middleware mengizinkan role "USER" untuk mengakses metode GET. Berbeda dengan metode POST yang diblokir untuk pengguna biasa, metode ini bersifat publik atau terbuka bagi pengguna terdaftar untuk melihat koleksi perpustakaan.
- Implementasi Pagination: Respons API tidak hanya mengembalikan array data mentah, melainkan menyertakan metadata halaman (Pagination). Terlihat adanya properti "page": 1, "limit": 3, dan "total": 3.
    - Fitur ini sangat krusial untuk performa aplikasi. Jika jumlah buku mencapai ribuan, sistem tidak akan membebani *bandwidth* dengan mengirimkan semua data sekaligus, melainkan membaginya per halaman. Dalam pengujian ini, sistem berhasil mendeteksi total 3 buku yang tersimpan di database.
- Struktur Data Konsisten: Pada array data, objek buku ditampilkan lengkap dengan relasi penulisnya (authorId). Data yang ditampilkan (seperti "The Eminence in Shadow 1") konsisten dengan data yang sebelumnya diinput oleh Admin, memverifikasi bahwa proses pengambilan data (querying) dari database berjalan akurat.

## 6. Pengujian PUT

Berikut ini adalah judul buku sebelum diubah




Lalu sesudahnya seperti digambar di bawah ini:




Penjelasan:

Pengujian ini bertujuan untuk memverifikasi fitur manipulasi data, yaitu mengubah informasi buku yang sudah tersimpan. Fitur ini menggunakan metode HTTP PUT dan memanfaatkan *Dynamic Routing* untuk menargetkan data spesifik berdasarkan ID buku.

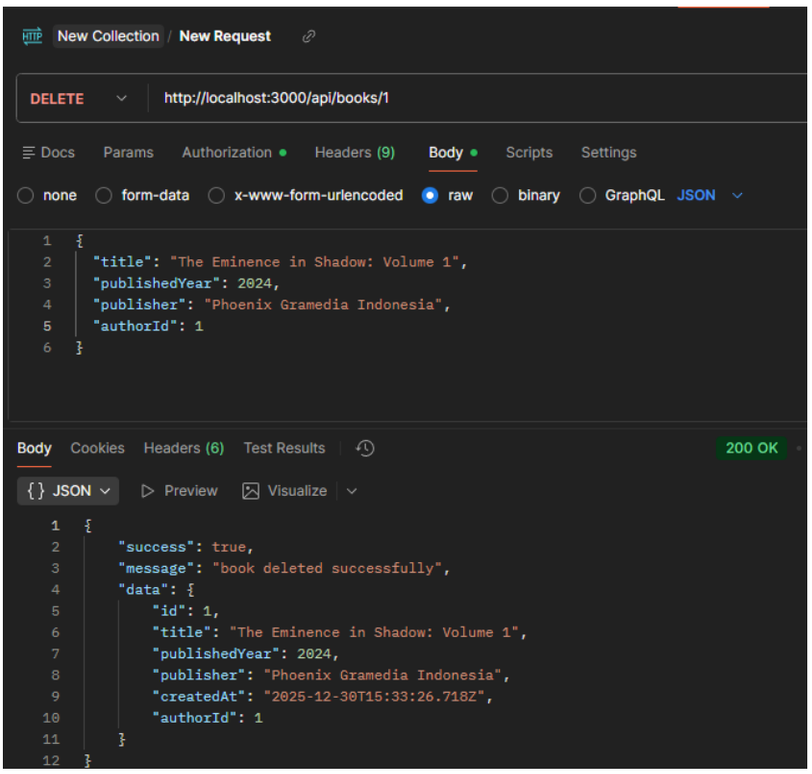
Detail Pengujian:

- Endpoint: http://localhost:3000/api/books/1 (Angka 1 adalah ID Buku yang akan diedit).
- METHOD: PUT
- Authorization: Bearer Token
- Payload (Body): Data JSON baru yang ingin disimpan (misalnya revisi judul atau tahun terbit).

Analisis Hasil:

- Penerapan Dynamic Routing ([id]): Permintaan dikirim ke URL yang menyertakan ID buku (/api/books/1). Next.js menangkap parameter 1 ini melalui mekanisme Dynamic Routes pada folder [id]. Sistem kemudian mencari data buku dengan ID tersebut di database. Jika ID tidak ditemukan, sistem dirancang untuk mengembalikan respon 404 Not Found.
- Validasi Hak Akses (Role Protection): Sebelum perubahan data diproses, kode program melakukan validasi ganda:
    - Autentikasi: Memastikan pengguna memiliki token yang valid.
    - Otorisasi: Memastikan atribut role pada token adalah "ADMIN". Jika pengguna biasa (User) mencoba mengakses endpoint ini, sistem akan menolak permintaan dengan status 403 Forbidden. Keberhasilan proses update pada gambar membuktikan bahwa token Admin yang digunakan memiliki izin yang sah.
- Integritas Pembaruan Data (Prisma Update): Server menerima data revisi dari client (misalnya perubahan judul menjadi "Volume 1"). Prisma ORM menjalankan perintah update secara aman ke PostgreSQL. Respon status 200 OK beserta data terbaru yang dikembalikan menunjukkan bahwa sinkronisasi antara request client dan data fisik di database telah berhasil dilakukan.

## 7. Pengujian DELETE




Penjelasan:

Tahap terakhir dalam pengujian manajemen data adalah memverifikasi fitur penghapusan (*Delete*). Fitur ini merupakan aksi destruktif yang bersifat permanen, sehingga memerlukan validasi hak akses yang ketat untuk mencegah hilangnya data penting oleh pihak yang tidak berwenang.

Detail Pengujian:

- Endpoint: http://localhost:3000/api/books/1
- Method: DELETE
- Authorization: Bearer Token (Token Admin).

Analisis Hasil:

- Validasi Otorisasi Tingkat Tinggi: Sistem menerima permintaan penghapusan pada ID buku 1. Sebelum mengeksekusi perintah, backend memvalidasi token JWT untuk memastikan pemohon memiliki role ADMIN. Sesuai dengan skenario keamanan yang dirancang, jika pengguna biasa (User) mencoba mengakses endpoint ini, sistem akan menolak akses tersebut. Keberhasilan operasi ini membuktikan bahwa mekanisme proteksi rute (Route Protection) berfungsi dengan baik.
- Penghapusan Data Permanen (Prisma Client): Sistem menggunakan perintah prisma.book.delete untuk mencari dan menghapus baris data yang sesuai

dengan ID yang diberikan. Berbeda dengan fitur Update yang memodifikasi nilai, fitur ini menghilangkan entitas buku beserta relasinya dari tabel basis data.

- Respon Balikan Data (Return Object): Server mengembalikan status HTTP 200 OK. Menariknya, respons JSON tetap mengembalikan objek data buku yang baru saja dihapus ("The Eminence in Shadow..."). Hal ini merupakan perilaku standar Prisma ORM yang berguna sebagai konfirmasi akhir kepada client mengenai detail data apa yang telah dihapus dari sistem.

# BAB III

## PENUTUP

Berdasarkan serangkaian pengujian yang telah dilakukan menggunakan REST Client (Postman), dapat disimpulkan bahwa API yang dibangun telah memenuhi seluruh spesifikasi teknis yang diminta:

1. Fungsionalitas CRUD: Sistem berhasil menangani operasi Create, Read, Update, dan Delete pada entitas Buku dan Penulis dengan alur data yang valid.
2. Keamanan (Security Layer):
   - Autentikasi: Implementasi JWT berhasil mengamankan akses, di mana pengguna tanpa token tidak dapat mengakses sumber daya yang dilindungi.
   - Otorisasi (RBAC): Pemisahan peran antara Admin dan User berjalan efektif. Admin memiliki kontrol penuh (CRUD), sedangkan User dibatasi hanya pada akses baca (Read-only).
3. Integritas Data: Validasi tipe data dan relasi antar tabel (Foreign Key) berhasil mencegah masuknya data sampah (garbage data) ke dalam sistem.

Dengan demikian, backend aplikasi ini dinyatakan siap untuk diintegrasikan dengan sisi frontend atau digunakan sebagai layanan mikro (microservice) yang mandiri.