LAPORAN PROJECT

Sistem ManajemenTugas/To-do

Laporan ini dibuat untuk memenuhi nilai tugas mata kuliah Pemrograman API

Dosen pengampu :

Saiful Nur Budiman, M.Kom

Disusun oleh :

Kelompok 6

1. **Dewi Nawang Wulandhary 23104410015**
2. **Ilana Rahmanisa Kholifatul Jannah 23104410002**
3. **Muchamad Choirul Ulum 23104410043**
4. **Aditya Putra Pratama 23104410033**
5. **Septian Abriyanto 23104410014**

PROGRAM STUDI TEKNIK INFORMATIKA

FAKULTAS TEKNIK DAN INFORMATIKA

UNIVERSITAS ISLAM BALITAR

2025

SETUP DATABASE DAN PRISMA

.env

```
DATABASE_URL="postgresql://postgres:baksoayam@localhost:5432/db_nextjs"
JWT_SECRET="kelompok06"
```

- **DATABASE_URL**: alamat koneksi ke database PostgreSQL yang digunakan aplikasi. Formatnya berisi username, password, host, port, dan nama database.
- **JWT_SECRET**: kunci rahasia yang dipakai untuk menandatangani dan memverifikasi token JWT agar proses autentikasi aman.

Skema model data

```
generator client {
  provider = "prisma-client-js"
  binaryTargets = ["native","rhel-openssl-3.0.x"]
}
datasource db {
 provider = "postgresql"
}
```

- **generator client**: mengatur Prisma Client yang akan dibuat, termasuk provider (prisma-client-js) dan target binary yang perlu didukung (native dan rhel-openssl-3.0.x).
- **datasource db**: menentukan sumber data utama yang digunakan Prisma, yaitu database dengan provider **PostgreSQL**.

Model user

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

Model **User** mendefinisikan struktur tabel pengguna, berisi:

- **id**: Primary key, bertambah otomatis.
- **name**: Nama pengguna.
- **email**: Harus unik.
- **password**: Menyimpan password yang sudah di-hash.
- **role**: Peran user, default-nya "USER".
- **createdAt**: Tanggal dibuatnya data, otomatis terisi waktu saat dibuat.

Model Katagori

```
model Category{
  id Int  @id @default(autoincrement())
  name String
  tasks Task[]
}
```

Model **Category** merepresentasikan tabel kategori yang berisi:

- **id**: Primary key, bertambah otomatis.
- **name**: Nama kategorinya.
- **tasks**: Relasi ke banyak **Task**, artinya satu kategori dapat memiliki banyak tugas.

Model Task

```
model Task{
  id Int @id @default(autoincrement())
  title String
  status String
  description String
  createdAt DateTime @default(now())

  //  Foreign Key(kolom dari id Kategory)
 categoryId Int

  //definisi relasi(prisma level)
  category Category @relation(fields: [categoryId], references: [id])
}
```

Model **Task** merepresentasikan data tugas, berisi:

- **id**: Primary key otomatis.
- **title**, **status**, **description**: Informasi tentang tugas.
- **createdAt**: Waktu dibuatnya tugas.
- **categoryId**: Foreign key yang menunjuk ke **Category**.
- **category**: Relasi Prisma yang menghubungkan setiap Task ke satu Category.

Tabel user

```
-- CreateTable
CREATE TABLE "User" (
    "id" SERIAL NOT NULL,
    "name" TEXT NOT NULL,
    "email" TEXT NOT NULL,
    "password" TEXT NOT NULL,
    "createdAt" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,
```

```
    CONSTRAINT "User_pkey" PRIMARY KEY ("id")
);

-- CreateIndex
CREATE UNIQUE INDEX "User_email_key" ON "User"("email");
```

- Perintah **CREATE TABLE "User"** membuat tabel *User* dengan kolom id (auto increment), name, email, password, dan createdAt.
- **PRIMARY KEY ("id")** menetapkan kolom id sebagai kunci utama.
- **CREATE UNIQUE INDEX "User_email_key"** memastikan nilai email harus unik (tidak boleh ada dua akun dengan email yang sama).

```
-- AlterTable
ALTER TABLE "User" ADD COLUMN     "role" TEXT NOT NULL DEFAULT 'USER';
```

Perintah tersebut menambahkan kolom **role** ke tabel **User**, dengan tipe **TEXT**, wajib diisi (**NOT NULL**), dan nilai default-nya adalah **"USER"**.

Tabel Item

```
-- CreateTable
CREATE TABLE "Category" (
    "id" SERIAL NOT NULL,
    "name" TEXT NOT NULL,

    CONSTRAINT "Category_pkey" PRIMARY KEY ("id")
);

-- CreateTable
CREATE TABLE "Task" (
    "id" SERIAL NOT NULL,
    "title" TEXT NOT NULL,
    "status" TEXT NOT NULL,
    "description" TEXT NOT NULL,
    "createdAt" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT "Task_pkey" PRIMARY KEY ("id")
);
```

- **Tabel Category** dibuat dengan kolom **id** (auto increment, primary key) dan **name**.
- **Tabel Task** dibuat dengan kolom **id**, **title**, **status**, **description**, dan **createdAt** (otomatis berisi waktu saat dibuat), dengan **id** sebagai primary key.

```
-- AlterTable
ALTER TABLE "Task" ADD COLUMN     "categoryId" INTEGER NOT NULL;

-- AddForeignKey
ALTER TABLE "Task" ADD CONSTRAINT "Task_categoryId_fkey" FOREIGN KEY
("categoryId") REFERENCES "Category"("id") ON DELETE RESTRICT ON UPDATE
CASCADE;
```

- **ALTER TABLE ... ADD COLUMN "categoryId"** menambahkan kolom *categoryId* ke tabel **Task** sebagai penanda kategori.
- **ADD FOREIGN KEY** menghubungkan *categoryId* pada **Task** ke kolom **id** di tabel **Category**, sehingga setiap Task harus memiliki Category yang valid.

MIDDLEWARE

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

export async function middleware(request) {
    const{pathname} = request.nextUrl;
```

Kode tersebut mengimpor **NextResponse** untuk mengirim respons dari middleware dan **jwtVerify** untuk memverifikasi token JWT. Pada fungsi middleware, baris const { pathname } = request.nextUrl; digunakan untuk mengambil bagian **path URL** yang sedang diakses pengguna.

```
    //1.Public : /api/auth/*
    if(pathname.startsWith("/api/auth")){
        return NextResponse.next();
    }
```

Kode tersebut memeriksa apakah URL yang diakses dimulai dengan **/api/auth**. Jika iya, berarti endpoint tersebut **bersifat publik**, sehingga middleware mengizinkan permintaan lanjut tanpa pemeriksaan apa pun dengan NextResponse.next().

```
    //2. cek header token
    const authHeader = request.headers.get("Authorization");

    if (!authHeader || !authHeader.startsWith("Bearer")){
        return NextResponse.json(
            {success: false, error: "Unathorized", code: 401},
            {status: 401}
        );
    }
    const token = authHeader.split(" ")[1]; //ambil string setelah "bearer"

    try{
```

Kode tersebut mengambil header **Authorization** dan memastikan header tersebut ada serta menggunakan format **Bearer <token>**. Jika tidak sesuai, permintaan ditolak dengan status 401. Jika valid, kode mengambil token-nya dengan memisahkan teks setelah kata *Bearer*. Kemudian blok try digunakan untuk memulai proses verifikasi token.

```
        //2.verfikasi token
        const secret = new TextEncoder().encode(process.env.JWT_SECRET);
        const {payload} = await jwtVerify(token, secret);
```

Kode tersebut melakukan **verifikasi token JWT**.

- secret dibuat dari nilai **JWT_SECRET** yang ada di environment, lalu diubah menjadi bentuk byte menggunakan TextEncoder().
- jwtVerify(token, secret) memeriksa apakah token valid dan ditandatangani dengan secret yang benar. Jika valid, isi token (payload) berhasil diambil dan disimpan dalam variabel **payload**.

```
        //admin only delete produk
        if (pathname.startsWith("/api/tasks")&& request.method === "DELETE"){
            if (payload.role !== "ADMIN"){
                return NextResponse.json({
                    success: false,
                    error: "Delete only untuk Admin",
                    code: 403
                }, {status: 403});
            }
        }
```

Kode tersebut membatasi akses **DELETE /api/tasks** hanya untuk user dengan role **ADMIN**. Jika endpoint yang diakses adalah /api/tasks dengan metode **DELETE** dan role di dalam payload token **bukan ADMIN**, maka permintaan ditolak dengan status **403 (Forbidden)** dan pesan bahwa delete hanya boleh dilakukan oleh admin.

```
        //3. Admin only: api.users/*
        if(pathname.startsWith("/api/users")&& payload.role !=="ADMIN"){
        return NextResponse.json(
            {success: false, error: "Akses hanya untuk Admin", code: 403},
            {status: 403}
        );
        }
```

Kode tersebut memeriksa apakah pengguna mengakses endpoint yang dimulai dengan **/api/users**. Jika iya, tetapi **role** di dalam token **bukan ADMIN**, maka akses ditolak dan server mengembalikan respons **403 (Forbidden)** dengan pesan bahwa hanya admin yang boleh mengakses endpoint tersebut.

```
        //3. Admin only: api.categories/*
        if(pathname.startsWith("/api/categories")&& payload.role !=="ADMIN"){
        return NextResponse.json(
            {success: false, error: "Akses hanya untuk Admin", code: 403},
            {status: 403}
        );
        }
```

Kode tersebut memeriksa apakah permintaan ditujukan ke endpoint **/api/categories**. Jika ya, tetapi **role** dalam token bukan **ADMIN**, maka akses ditolak dan server mengembalikan respons **403 (Forbidden)** dengan pesan bahwa hanya admin yang boleh mengakses kategori.

```
        //4. Users: /api/tasks/*
        if(pathname.startsWith("/api/tasks")&& !payload.id){
            return NextResponse.json(
                {success: false, error: "Invalid User Token", code: 403},
                {status: 403}
            );
        }
```

Kode tersebut memastikan bahwa setiap permintaan ke endpoint **/api/tasks** hanya boleh dilakukan oleh user yang memiliki **ID valid** di dalam token. Jika payload.id tidak ada (token tidak valid atau tidak berisi data user), maka akses ditolak dan server mengembalikan **403 (Forbidden)** dengan pesan "Invalid User Token".

```
        //5.lolos
        return NextResponse.next();
    }catch(error){
        //6. Jika token salah/expired
        return NextResponse.json(
            {success:false, error: "Invalid Token", code: 401},
            {status: 401}
        );
    }
}
```

- **return NextResponse.next()** → Jika semua pengecekan berhasil, permintaan diizinkan untuk dilanjutkan.
- **catch(error)** → Jika terjadi kesalahan saat verifikasi token (misalnya token salah atau sudah kadaluarsa), middleware membalas dengan status **401 (Unauthorized)** dan pesan **"Invalid Token"**.

```
export const config = {
    matcher: ["/api/users/:path*", "/api/tasks/:path*",
"/api/categories/:path*"],
};
```

Konfigurasi **matcher** menentukan rute mana saja yang akan diproses oleh middleware.

Dalam hal ini, middleware hanya dijalankan untuk URL yang dimulai dengan:

- **/api/users/**
- **/api/tasks/**
- **/api/categories/**

Semua path setelahnya (:path*) juga ikut ter-cover.

CRUD PROTECTED API

Users

- Route.js

```
import {prisma} from "@/lib/prisma";
import { error } from "console";
import { NextResponse } from "next/server";
import {email, success, z} from "zod";
```

- **prisma** → Menghubungkan ke database menggunakan Prisma Client.
- **error dari console** → Untuk mencatat error di console (jarang dipakai).
- **NextResponse** → Mengirim respons HTTP pada route handler Next.js.
- **email, success, z dari Zod** → Dipakai untuk membuat dan menjalankan validasi data (misalnya format email, cek hasil validasi, dan membuat schema).

```
const UserSchema = z.object({
    name: z.string().min(3,{message: "Nama minimal harus 3 karakter"}),
    email: z.string().email({message: "Format email tidak valid"}),
    password: z.string().min(8,{message: "Password minimal harus 8
karakter"}),
});
```

Kode tersebut membuat **schema validasi** menggunakan **Zod** untuk memastikan data user yang masuk sudah benar. **UserSchema** berisi aturan:

- **name**: Harus berupa string, minimal 3 karakter.
- **email**: Harus berupa email dengan format valid.
- **password**: Harus berupa string dengan panjang minimal 8 karakter.

Schema ini digunakan untuk memvalidasi input sebelum data disimpan ke database.

```
//GET : Ambil semua data user
export async function GET() {
    try{
    const users =await prisma.user.findMany();
    return NextResponse.json({
        success: true,
        message: "All users fetched",
        data: users
    });
}catch(error){
    return NextResponse.json({
        success: false,
        error: "Server Error",
        code: 500
    }, {status: 500});
}
}
```

- Fungsi **GET** ini mengambil semua data user dari database menggunakan prisma.user.findMany().
- Jika berhasil, server mengembalikan respons **success** berisi daftar seluruh user.
- Jika terjadi error, fungsi menangkapnya dan mengirim respons **500 (Server Error)**.

```
//POST : Tambah user baru
export async function POST(request) {
    try{
    const body =await request.json();
    const validation = UserSchema.safeParse(body);
    if(!validation.success){
        return NextResponse.json({
            message: "Input Tidak Valid",
            error: validation.error.flatten().fieldErrors
        },
    {status: 400});
    }
    const newUser =await prisma.user.create({
        data: body
    });
    return NextResponse.json({
        success: true,
        message: "User created",
        data: newUser
    },{status: 200});
}catch(error){
    console.error("ERROR POST User: ", error);
    NextResponse.json({message: "Server Error"}, {status: 500});
}
```

1. **Mengambil data dari request** → request.json().
2. **Validasi input** menggunakan UserSchema.safeParse().
   Jika tidak valid, server mengirim respons **400** dengan detail error.
3. **Membuat user baru** di database lewat prisma.user.create({ data: body }).
4. Jika berhasil, mengembalikan respons sukses beserta data user.
5. Jika terjadi error, ditangani di blok catch dan mengirim respons **500 (Server Error)**.

- **[id]route.js**

```
import { NextResponse } from "next/server";
import { prisma } from "@/lib/prisma";
import { success } from "zod";
```

- **NextResponse** → Digunakan untuk mengirim respons dalam route handler Next.js.
- **prisma** → Mengakses database melalui Prisma Client.
- **success dari zod** → Biasanya digunakan sebagai helper dari Zod, tetapi pada kode ini sebenarnya **tidak diperlukan** jika tidak digunakan dalam validasi.

```
export async function GET(request, {params}) {
    try{
        const resolvedParams = await params;
        const id = parseInt(resolvedParams.id);

        if(isNaN(id)){
            return NextResponse.json(
                {message: "ID tidak valid"}, {status: 400}
            );
        }
```

Kode **GET** ini digunakan untuk mengambil data user berdasarkan **ID** dari URL.

- params berisi parameter URL, lalu diambil id-nya.
- parseInt() mengubah ID dari string menjadi angka.
- Jika ID bukan angka (isNaN(id)), maka server mengembalikan respons **400 (Bad Request)** dengan pesan **"ID tidak valid"**.

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

        if(!user){
            return NextResponse.json(
                {message: "User tidak ditemukan"},
                {status: 400}
            );
        }

        return NextResponse.json({
            success: true,
            message: "User detail fetched",
            data: user
        }, {status: 200});

    }catch(error){
        console.error("Error GET User:", error);
        return NextResponse.json(
            {message: "Error Server"},{status: 500}
        )
    }
}
```

Kode tersebut mencari user berdasarkan **ID** menggunakan prisma.user.findUnique().

- Jika user **tidak ditemukan**, server mengembalikan respons **400** dengan pesan "User tidak ditemukan".
- Jika user **ditemukan**, data user dikirim kembali dengan status **200**.
- Jika terjadi error di proses, blok catch mengirimkan respons **500 (Error Server)** dan mencatat error di console.

```
export async function PUT(request,{params}) {
    try{
        //ambil id dari params
        const resolvedParams = await params;
        const id = parseInt(resolvedParams.id);

        //ambil data dari body
        const data = await request.json();

        //update user
        const updateUser = await prisma.user.update({
            where: {id: id},
            data:{
                name: data.name,
                email: data.email,
                password: data.password
            }
        });
        return NextResponse.json(
            {
                success: true,
                message: "Update berhasil",
                data: updateUser
            },
            {status: 200}
        );
    }catch(error){
        console.error("Error PUT/api/users/[id]:", error);

        return NextResponse.json(
            {
                message: "Terjadi kesalahan saat update user",
                error: error.message
            },
            {status: 500}
        );
    }
}
```

Fungsi **PUT** ini digunakan untuk mengubah data user berdasarkan **ID** yang diambil dari parameter URL. Setelah ID diubah menjadi angka, server

mengambil data baru dari body request. Prisma kemudian menjalankan proses update pada tabel user dengan mengganti nama, email, dan password sesuai input baru. Jika proses berhasil, server mengirimkan respons sukses berisi data user yang telah diperbarui. Namun bila terjadi kesalahan, blok **catch** menangkap error dan mengembalikan respons **500** dengan pesan kegagalan.

```
export async function DELETE(request, {params}) {
    try{
        //Ambil ID dari parameter URL
        const resolvedParams = await params;
        const id = parseInt(resolvedParams.id);

        //hapus data user
        const deletedUser = await prisma.user.delete({
            where: {id: id}
        });

        return NextResponse.json(
            {
                success: true,
                message: "User berhasil dihapus",
                data: deletedUser
            },
            {status: 200}
        );
    }catch(error){
        console.error("Error DELETE /api/users/[id]:", error)

        return NextResponse.json(
            {
                message: "Terjadi keslahan saat menghapus user",
                error: error.message
            },
            {status : 500}
        );
    }
}
```

Fungsi **DELETE** ini bekerja untuk menghapus user berdasarkan **ID** yang diperoleh dari parameter URL. Setelah ID diubah menjadi angka, Prisma menjalankan perintah delete() untuk menghapus data user yang sesuai di database. Jika proses berhasil, server mengirimkan respons sukses berisi pesan bahwa user telah dihapus beserta data user yang terhapus. Apabila terjadi kesalahan, blok **catch** menangkap error tersebut, mencatatnya di console, lalu mengembalikan respons **500** dengan informasi bahwa terjadi masalah saat proses penghapusan.

Tasks

- Route.js

```
import { NextResponse } from "next/server";
import {prisma} from "@/lib/prisma";
import { connect } from "http2";
import { success } from "zod";

export async function GET() {
    try{
        const tasks = await prisma.task.findMany({
            include: {category: true}
        });
        return NextResponse.json({
            success: true,
            message: "Tasks fetched",
            data: tasks
        }, {status: 201});
    }catch(error){
        console.error("Server error", error);
        return NextResponse.json({error: "Internal Server Error"},
{status: 500});
    }
```

Fungsi **GET** ini mengambil semua data **Task** dari database menggunakan prisma.task.findMany(). Properti include: { category: true } digunakan untuk menampilkan data kategori yang terkait dengan setiap task. Jika berhasil, server mengembalikan respons sukses dengan daftar task beserta kategorinya. Jika terjadi error pada proses pengambilan data, blok **catch** menangkap error tersebut dan mengirimkan respons **500 (Internal Server Error)**.

```
export async function POST (request) {
    try{
        const data = await request.json();
        const newTask = await prisma.task.create({
            data:{
                title: data.title,
                status: data.status,
                description: data.description,
                category: {
                    connect: {
                        id: data.categoryId // id dari kategori yang
sudah dibuat
                    }
                }
            },
        });
        return NextResponse.json({
            success: true,
```

```
            message: "Tasks created",
            data: newTask
        }, {status: 201});
    }catch(error){
        console.error("Server error", error);
        return NextResponse.json({error: "Internal Server Error"},
{status: 500});
    }
```

Fungsi **POST** ini digunakan untuk membuat task baru. Data dari request dibaca lalu diteruskan ke prisma.task.create(). Kolom **title**, **status**, dan **description** diisi sesuai input, sedangkan kategori task ditentukan dengan menghubungkan (connect) **categoryId** yang dikirim dari client. Jika task berhasil dibuat, server mengembalikan respons sukses dengan status 201. Jika terjadi masalah selama proses, error ditangkap dan server mengirim respons **500 (Internal Server Error)**.

- [id]route.js

```
export async function PUT(request, {params}) {
    try{
        const resolvedParams= await params;
        const idTask = parseInt(resolvedParams.id);

        const data = await request.json();
```

Kode tersebut berada dalam fungsi **PUT** untuk meng-update data Task.

- params diambil dari URL, lalu diselesaikan (resolvedParams) dan diubah menjadi angka dengan parseInt() untuk mendapatkan **idTask**.
- Setelah itu, server membaca body request menggunakan request.json() untuk mendapatkan data baru yang akan digunakan dalam proses update task.

```
        //1.siapkan objek data untuk update
        let updateData = {
            title: data.title,
            status: data.status,
            description: data.description
        };
```

Kode tersebut membuat objek **updateData** yang berisi data baru untuk memperbarui sebuah task. Nilai **title**, **status**, dan **description** diambil dari input pengguna (*data*) dan disiapkan untuk dikirim ke database dalam proses update. Objek ini memastikan hanya field tertentu yang akan diperbarui pada task tersebut.

```
        //2. logika disconnect
```

```
        if(data.hapusKategori==true){
            updateData.category={
                disconnect: true
            };
        }
        else if(data.categoryId){
            updateData.category={
                connect:{
                    id: data.categoryId
                }
```

Kode tersebut mengatur apakah kategori pada task akan dihapus atau diganti.

- Jika **hapusKategori == true**, maka task akan **memutus hubungan** dengan kategori menggunakan disconnect: true.
- Jika **categoryId** diberikan, maka task akan **menghubungkan** diri ke kategori baru menggunakan connect: { id: data.categoryId }. Bagian ini memastikan relasi kategori dapat dihapus atau diganti sesuai input dari pengguna.

```
        //3.update data
        const updateTask = await prisma.task.update({
            where: {id: idTask},
            data: updateData,
            include: {
                category: true
            }
        });
        return NextResponse.json(
            {
                success: true,
                message: "Tugas berhasil diupdate",
                data: updateTask
            },
            {status: 200}
        );
    }catch(error){
        console.error("Server Error", error);
        return NextResponse.json({error: "Internal Server Error"},
{status:500})
    }
```

Kode tersebut melakukan proses **update task** di database menggunakan prisma.task.update().

Task dicari berdasarkan **idTask**, lalu diubah sesuai isi updateData, dan data kategori juga ikut ditampilkan melalui include: { category: true }. Jika update berhasil, server mengembalikan respons sukses dengan status **200** dan data task terbaru. Namun, jika terjadi error, blok **catch** menangkapnya dan

mengirimkan respons **500 (Internal Server Error)** sambil mencatat error ke console.

```
export async function DELETE(request, {params}) {
    try{
        const resolvedParams= await params;
        const id = parseInt(resolvedParams.id);
        // const id =parseInt(params.id);
        const deletedTask = await prisma.task.delete({
            where: {id}
        });

        return NextResponse.json({
            success: true,
            message: "List terhapus",
            data: deletedTask
        });
    }catch(error){
        console.error("Error DELETE /api/tasks/[id]:",error);

        return NextResponse.json({
            success: false,
            error: "Gagal menghapus list",
            code: 500
        }, {status: 500});
    }
```

Fungsi **DELETE** ini digunakan untuk menghapus task berdasarkan **ID** dari URL. Setelah ID diubah menjadi angka, Prisma menjalankan task.delete() untuk menghapus task yang sesuai di database. Jika penghapusan berhasil, server mengirim respons sukses berisi pesan bahwa task telah terhapus dan menampilkan data task yang dihapus. Jika terjadi kesalahan, error dicatat pada console dan server membalas dengan respons **500**, menandakan bahwa penghapusan gagal dilakukan.

Categories

- Route.js

```
export async function GET() {
    try{
        const students = await prisma.student.findMany();

        return NextResponse.json(tasks, {status: 201});
    }catch(error){
        console.error("Server error", error);
        return NextResponse.json({error: "Internal Server Errror"},
{status: 500});
    }
```

Fungsi **GET** ini mencoba mengambil semua data dari tabel **student** menggunakan prisma.student.findMany(). Jika berhasil, seharusnya data **students** dikembalikan dalam bentuk JSON. Namun, terdapat **kesalahan kode** karena respons mengembalikan variabel **tasks**, yang tidak didefinisikan dalam fungsi tersebut. Jika terjadi error saat mengambil data, server mengembalikan respons **500 (Internal Server Error)**.

```
export async function POST(request) {
    try{
        const data = await request.json();
        const newCategory = await prisma.category.create({
            data:{
                name: data.name
            },
        });
        return NextResponse.json(newCategory);
    }catch(error){
        console.error("Server error", error);
        return NextResponse.json({error:"Internal Server Error"},
{status: 500});
    }
```

Fungsi **POST** ini digunakan untuk menambahkan kategori baru. Data dari body request diambil melalui request.json(), lalu Prisma menjalankan category.create() untuk menyimpan kategori baru ke database dengan field **name** saja. Jika berhasil, server mengirim respons berisi data kategori yang baru dibuat. Jika terjadi kesalahan, error dicatat dan server mengembalikan respons **500 (Internal Server Error)**.

Login

- Route.js

```
export async function POST(request) {
    try {
        const { email, password } = await request.json();

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

Kode tersebut menangani proses login dengan metode POST. Pertama, server mengambil email dan password dari body request. Lalu server mencari data user berdasarkan email di database menggunakan Prisma. Jika user tidak ditemukan, sistem langsung mengembalikan respons error dengan status 401

yang berarti kredensial tidak valid. Tahap ini penting untuk memastikan bahwa hanya pengguna yang terdaftar yang dapat melanjutkan proses verifikasi password dan login, sehingga akses tetap aman dan tidak diberikan kepada email yang tidak ada dalam database.

```
        // Cek password
        const valid = await bcrypt.compare(password, user.password);
        if (!valid) return NextResponse.json({ error: "Invalid
credentials" }, { status: 401 });

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

Kode ini melakukan verifikasi password saat login. Fungsi bcrypt.compare() membandingkan password yang dikirim pengguna dengan password yang tersimpan di database (yang sudah di-hash). Jika password tidak cocok, server mengembalikan respons **401 (Invalid credentials)**. Jika cocok, server menyiapkan kunci rahasia JWT dengan mengambil nilai **JWT_SECRET** dari environment dan mengubahnya menjadi format byte menggunakan TextEncoder(). Kunci ini nantinya digunakan untuk membuat atau memverifikasi token JWT agar autentikasi tetap aman.

```
        // Access Token – expired cepat
        const accessToken = await new SignJWT({
            id: user.id,
            email: user.email,
            role: user.role,
        })
            .setProtectedHeader({ alg: "HS256" })
            .setExpirationTime("1h")
            .sign(secret);
```

Kode ini membuat **JWT (JSON Web Token)** untuk autentikasi user. Token berisi informasi penting user seperti **id, email, dan role**. setProtectedHeader({ alg: "HS256" }) menentukan algoritma tanda tangan token menggunakan HMAC SHA-256. setExpirationTime("1h") membuat token hanya berlaku selama satu jam. Terakhir, .sign(secret) menandatangani token menggunakan secret yang sudah diubah menjadi format byte, sehingga token dapat diverifikasi keasliannya di server dan memastikan akses aman bagi user yang sah.

```
        // Refresh Token – expired lama
        const refreshToken = await new SignJWT({
            id: user.id,
        })
            .setProtectedHeader({ alg: "HS256" })
            .setExpirationTime("7d")
```

```
            .sign(secret);
```

Kode ini membuat **refresh token** untuk user yang sudah login. Refresh token hanya berisi **id user** dan berfungsi untuk memperbarui akses token ketika sudah kadaluarsa. Header token diatur dengan algoritma **HS256**, sama seperti akses token. Expiration time diatur **7 hari**, lebih panjang dari akses token agar user tidak perlu login ulang terlalu sering. Token kemudian ditandatangani dengan secret server menggunakan .sign(secret), sehingga server bisa memverifikasi keasliannya saat digunakan untuk mendapatkan akses token baru.

```
        // Simpan refresh token di cookie HttpOnly
        const response = NextResponse.json({
            message: "Login Success",
            accessToken: accessToken,
        });
```

Kode tersebut membuat respons **JSON** untuk dikirim ke client setelah login berhasil. Objek JSON berisi pesan "Login Success" dan **accessToken** yang baru dibuat. NextResponse.json() digunakan untuk membungkus data ini menjadi respons HTTP yang valid, sehingga frontend dapat menerima token dan menyimpannya (misalnya di localStorage atau cookie) untuk digunakan dalam autentikasi saat mengakses endpoint yang membutuhkan hak akses user. Respons ini menandakan bahwa login berhasil dan user dapat mengakses sumber daya yang terlindungi.

```
        response.cookies.set("refresh_token", refreshToken, {
            httpOnly: true,
            secure: false, // ubah ke true kalau deploy
            path: "/",
            maxAge: 7 * 24 * 60 * 60,
        });

        return response;

    } catch (error) {
        console.error("LOGIN ERROR:", error);
        return NextResponse.json({ error: "Internal Server Error" }, {
status: 500 });
    }
}
```

Kode ini menyimpan **refresh token** di cookie dengan opsi **httpOnly** agar tidak bisa diakses JavaScript. Cookie berlaku selama 7 hari. Setelah itu, respons dikirim ke client. Jika terjadi error selama login, server menangkapnya dan mengembalikan respons **500 (Internal Server Error)**.

Register

- Route.js

```
export async function POST(request) {
    try{
        const{name, email, password,role} = await request.json();

        //1.Hash Password
        const hashedPassword = await bcrypt.hash(password, 10);

        //2.simpan ke DB
        const newUser = await prisma.user.create({
            data: {
                name,
                email,
                password: hashedPassword, //simpan yang sudah di-hash!
                role: role || "USER",
            },
        });

        return NextResponse.json({message: "User Created"}, {status:
201});
    }catch(error){
        console.error("Error GET User:", error);
        return NextResponse.json({message: "Error"},{status: 500});
    }
```

Fungsi **POST** ini digunakan untuk menambahkan user baru ke database. Pertama, server mengambil **name, email, password, dan role** dari body request. Password kemudian di-hash menggunakan **bcrypt** untuk keamanan. Setelah itu, Prisma menyimpan user baru ke database, dengan password yang sudah di-hash dan role default "USER" jika tidak diberikan. Jika berhasil, server mengirim respons **201 (Created)**. Jika terjadi error, blok **catch** menangkapnya dan mengembalikan respons **500 (Internal Server Error)** serta mencatat error di console.

REFRESH TOKEN

- **app/api/auth/login/route.js**

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

- **NextResponse** : digunakan untuk membalas request dalam route Next.js App Router.
- **prisma** : untuk query database.

- **bcrypt** : untuk membandingkan password yang diinput user dengan password yang sudah dihash dalam database.
- **SignJWT (jose)** : library untuk membuat JWT access token dan refresh token.

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

Perintah untuk mengambil field email dan password.

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

```
    if (!user) return NextResponse.json({ error: "Invalid credentials" }, { status: 401 });
```

Perintah untuk mencari user dalam db berdasarkan email. Apabila tidak ada makan akan mengirim respon error.

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

```
    if (!valid) return NextResponse.json({ error: "Invalid credentials" }, { status: 401 });
```

Perintah untuk membandingkan password menggunakan bcrypt dengan cara melakukan cek pada password yang dimasukkan oleh user apakah sama dengan password hashed di db. Apabila salah makan akan mengirim respon error.

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

Perintah untuk membuat secret key dimaan mengambil JWT_SECRET dari .env

```
const accessToken = await new SignJWT({
```

```
        id: user.id,
```

```
        email: user.email,
```

```
        role: user.role,
```

```
    })
```

```
        .setProtectedHeader({ alg: "HS256" })
```

```
        .setExpirationTime("1h")
```

```
        .sign(secret);
```

Perintah untuk mmebuat access token dengan masa expired cepat dengan masa satu jam dan ditandatangani dengan SECRET.

```
const refreshToken = await new SignJWT({
```

```
  id: user.id,
```

```
})
```

```
  .setProtectedHeader({ alg: "HS256" })
```

```
    .setExpirationTime("7d")
    .sign(secret);
```

Perintah untuk melakukan refresh token dengan masa expired lama dengan masa tujuh hari.

```
const response = NextResponse.json({
    message: "Login Success",
    accessToken: accessToken,
});
```

Perintah untuk membuat response JSON pada frontend apabila login success dengan disertai akses token.

```
response.cookies.set("refresh_token", refreshToken, {
    httpOnly: true,
    secure: false,
    path: "/",
    maxAge: 7 * 24 * 60 * 60,
});
```

Perintah untuk menyimpan refresh token ke cookie HttpOnly dengan nama cookie refresh_token dengan maxAge selama tujuh hari.

```
return response;
```

Perintah untuk mengirim JSON berupa respon Login Succes dan mengirim access token dan juga cookie refresh token.

```
 } catch (error) {
        console.error("LOGIN ERROR:", error);
        return NextResponse.json({ error: "Internal Server Error" }, { status: 500 });
    }
}
```

Perintah untuk error handling, dimana apabila terjadi error dalam proses maka akan mengirim respon Internal Server Error dengan status 500.

- **app/api/auth/refresh**

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

```
import { jwtVerify, SignJWT } from "jose";
```

Perintah untuk import library dimana jwtVerify untuk melakukan verifikasi token JWT dan SignJWT untuk membuat token JWT yang baru.

```
export async function POST(request) {
    try {
        const refreshToken = request.cookies.get("refresh_token")?.value;
```

Route akan dipanggil Ketika user mengirim methode POST request ke endpoint refresh token dan refresh token akan disimpan di cookie HttpOnly.

```
  if (!refreshToken) {
            return NextResponse.json({ error: "No refresh token" }, { status: 401 });
        }
```

Logika apabila tidak ada refresh token maka akan memberikan respon No refresh token dengan status 401.

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

Perintah untuk melakukan verifikasi token dan membuat token baru.

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

perintah untuk verifikasi refresh token, dimana apabila valid maka akan mengambil payload yang berisi id dan apabila tidak valid makan akan error.

```
const newAccessToken = await new SignJWT({
    id: payload.id,
})
    .setProtectedHeader({ alg: "HS256" })
    .setExpirationTime("1h")
    .sign(secret);
```

Perintah untuk membuat access token baru dengan melakukan payload token hanya menggunakan id, expired time tojen adalah satu jam dan token ditandatangani dengan secret yang sama.

```
return NextResponse.json({
    message: "Token refreshed",
    accessToken: newAccessToken,
});
```

Melakukan return pada user untuk menampilkam access token yang baru ke user.

```
} catch (error) {
    return NextResponse.json({ error: "Invalid Refresh Token" }, { status: 401 });
}
```

Jika token invalid maka akan mengirim respon Invalid Refresh Token dengan status 401.

PAGINATION

- **lib/db-helper.js**

```
export async function getPostsFromDB(limit, offset) {

  const totalCount = await prisma.task.count();
```

Melakukan export function getPostFromDB, dimana mengambil data student dari data base dengan pagination dengan limit adalah jumlah data per halaman dan offset adalah jumlah data yang dilewati. Lalu menghitung jumlah total baris di table student.

```
const posts = await prisma.task.findMany({
    take: limit, // Batas jumlah item
    skip: offset, // Lewati sejumlah item
    orderBy: {
      id: 'asc',
    },
  });
```

Query untuk mengambil data dimana take: limit yaitu mengambil banyak data, skip: offset melewati beberapa data dimana apabila melkaukan setting limit = 10 maka setelah 10 data yang ditampilkan akan menampilkan halaman selanjutnya. orderBy digunakan untuk melakukan pengurutan data berdasarkan id secara ascending agar data sellau konsisten.

```
console.log("Data returned:", posts.length);
```

Perintah untuk menampilkan jumah hasil query di konsol server.

```
return { posts, totalCount };
```

Perintah untuk melakukan return dua nilai yaitu post dimana merupakan data student yang diambil menggunakan limit dan offset dan totalCount yang merupakan total seluruh data di database.

- **app/api/pagination**

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

import { getPostsFromDB } from '@/lib/db-helper';

import library

export async function GET(request) {

  try {
```

```
    const { searchParams } = new URL(request.url);
    const limit = parseInt(searchParams.get("limit")) || 10;
    const page = parseInt(searchParams.get("page")) || 1;
```

Endpoint pagination yang dipanggil saat client mengirimkan methode GET dengan logika try yaitu (searchParams.get("limit") untuk mengambil jumlah data per halaman yaitu 10 dan (searchParams.get("page") untuk mengambil nomor halaman yaitu 1.

```
if (limit <= 0 || page <= 0) {
      return NextResponse.json({ message: "Invalid page or limit value" }, { status: 400 });
    }
```

Logika untuk melakukan validasi parameter yaitu apabila user memberikan nilai pada limit -2 atau page -1 maka akan mengirim erespon Invalid page or limit value dengan status 400.

```
const offset = (page - 1) * limit;
console.log("Offset / Skip:", offset);
```

Melakuakn perhitungan untuk offset dimana page 1 sama dengan offset = 0 dan page 2 sama dengan offset = limit.

```
const { posts, totalCount } = await getPostsFromDB(limit, offset);
```

Perintah untuk mengambil data dari dataase.

```
 const totalPages = Math.ceil(totalCount / limit);
```

Perinah untuk menghitung total halaman. Menggunakan Math.ceil agar nilai hasil dibulatkan ke atas.

```
return NextResponse.json({
  message: "Data fetched successfully",
  data: posts,
  pagination: {
    totalItems: totalCount,
    totalPages: totalPages,
    currentPage: page,
    itemsPerPage: limit,
    hasNextPage: page < totalPages,
    hasPrevPage: page > 1,
```

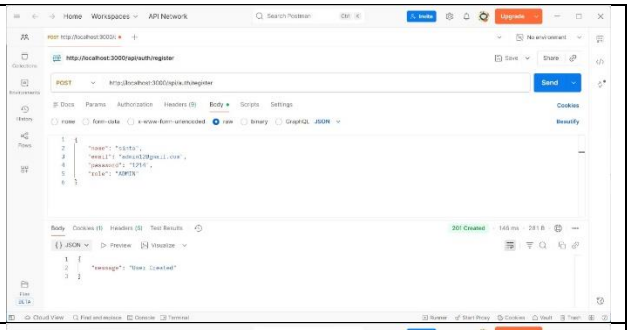
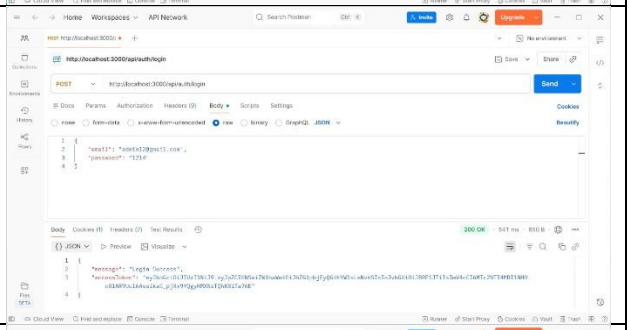
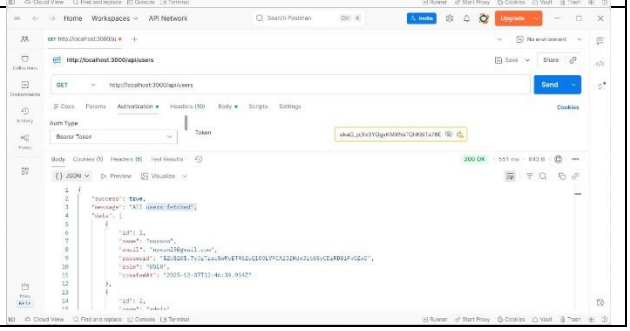
```
  },
});
```

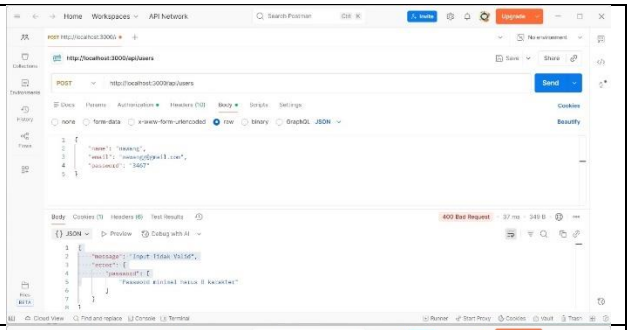
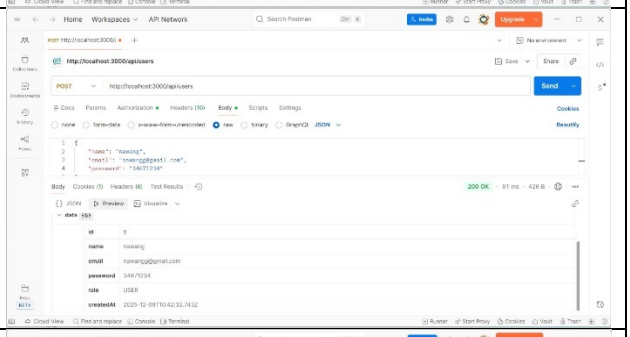
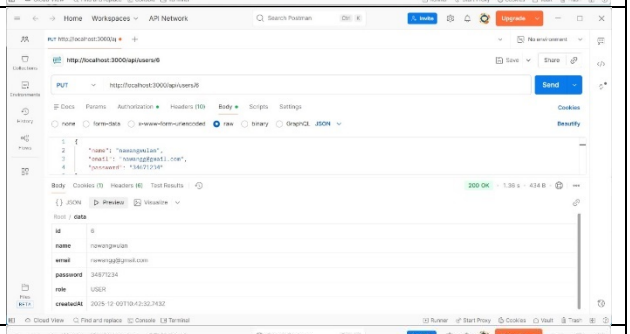
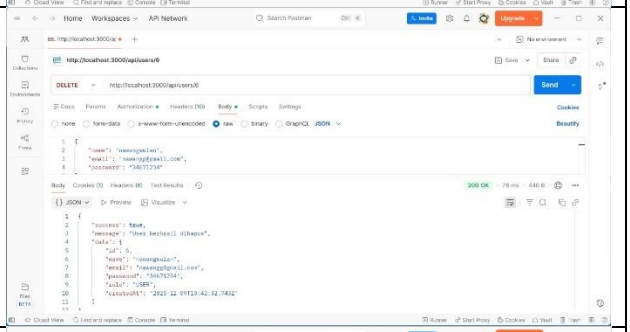
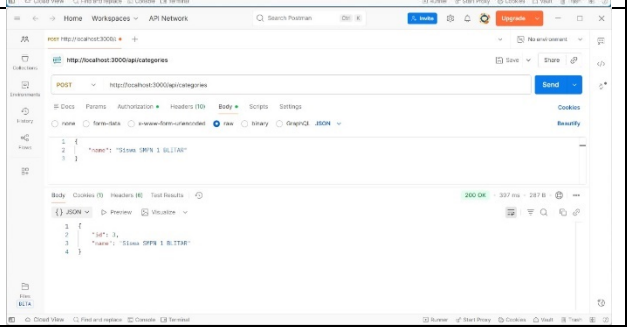
Merupakan data dari hasil query.

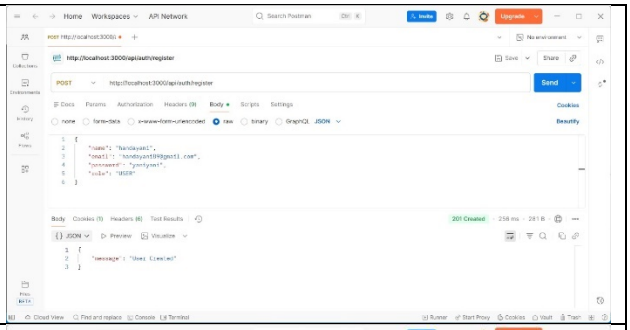
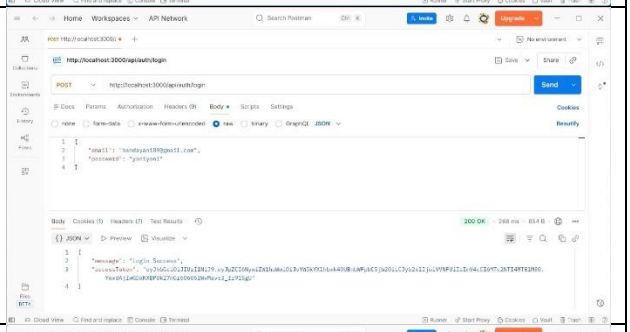
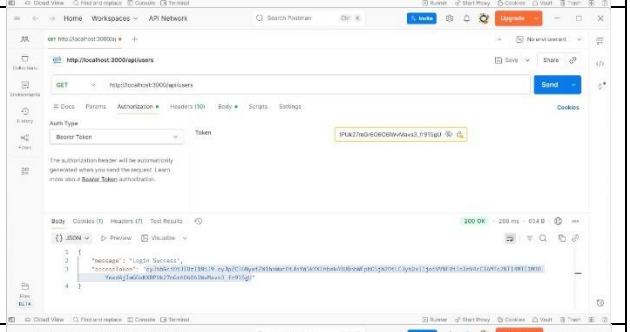
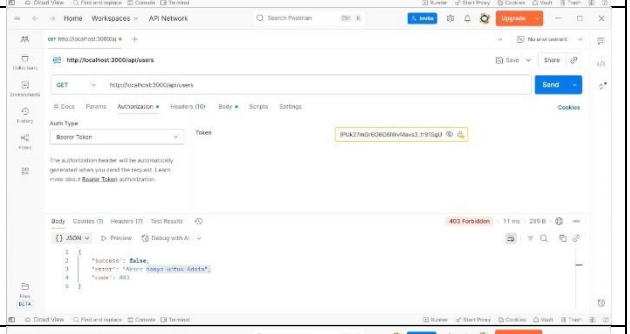
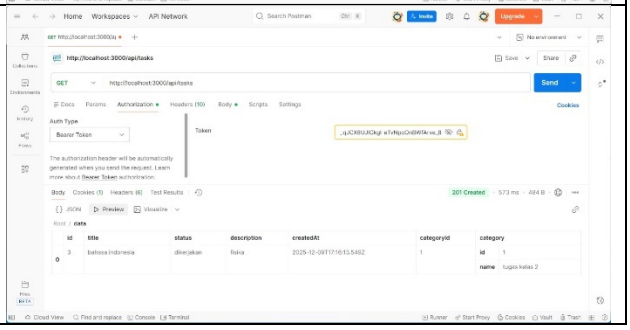
```
} catch (error) {
  console.error("API Error:", error);
  return NextResponse.json({ message: "Internal Server Error" }, { status: 500 });
}
```

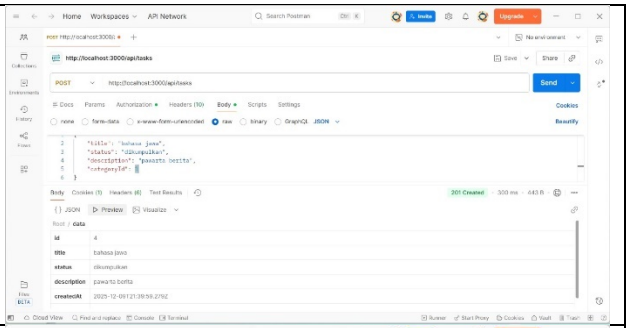
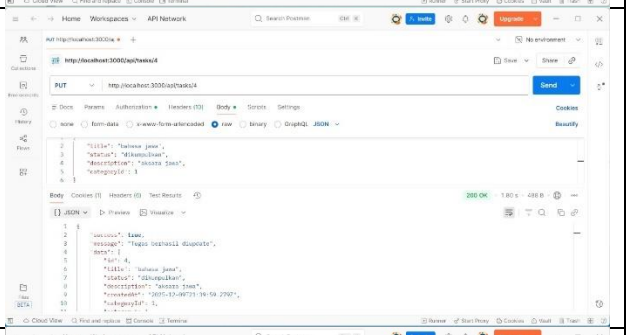
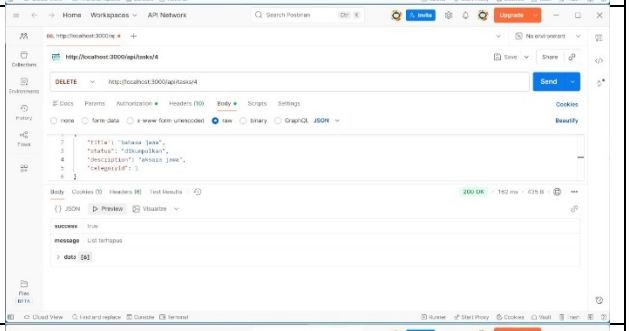
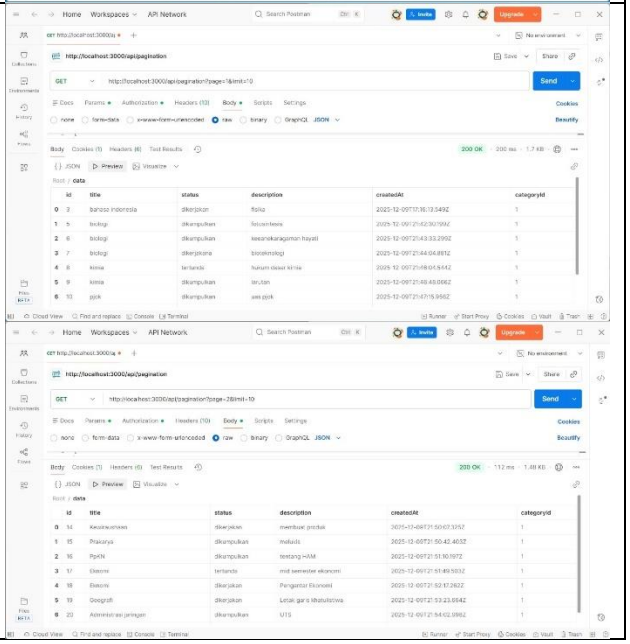
Error handling.

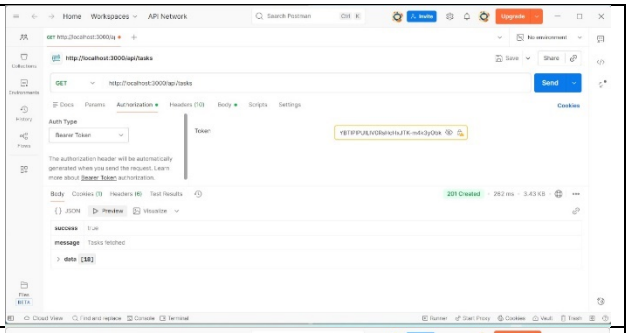
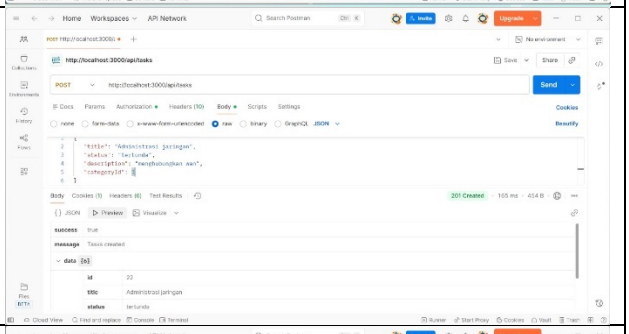
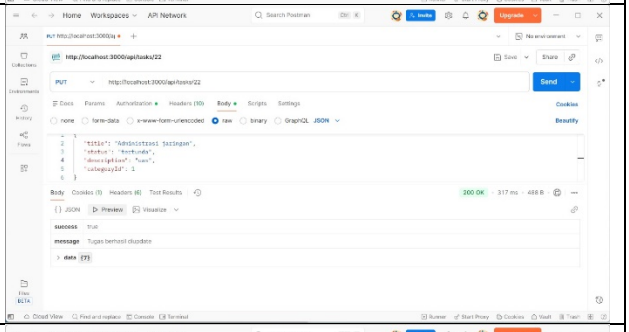
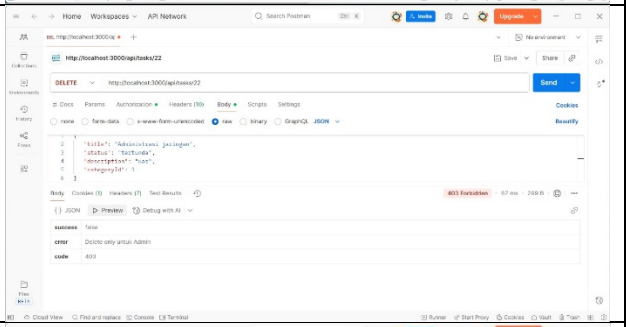
TESTING

auth/register admin	
auth/login admin	
get all users mode admin	

post users jika salah	
post users berhasil	
put update users	
delete data admin	
post categories admin	

auth/register user	
auth/login users	
masuk ke users mode user	
hasil	
get tugas admin	

post task admin	
put task admin	
delete	
pagination page 1 dan page 2	

get task user	
post users	
put	
delete	
Refresh token