wImplementasi Zero Trust Access Control pada Aplikasi Sederhana

kelompok 4
Luthfi Kurniawan (2201020013)
M. Afief Anugrah (2201020015)
Aditya Firmansyah (2201020018)
Halta Putra Ash Sidiq (2201020092)

Minggu 2 : Implementasi login + hashing (bcrypt)

1. Kode Implementasi: Fungsi Registrasi

Tampilkan kode fungsi postRegisterUsers di sini:

```
public async postRegisterUsers(
    req: Request,
    res: Response
  ): Promise<Response> {
    try {
      const { username, password, roleId, isActive } =
        req.body as createUserBody;

      if (!username || !password || !roleId) {
        return sendError(res, "Mohon isi username, password, dan role,", 400);
      }

      const cekUsername = await Users.findOne({ where: { username } });

      if (cekUsername) {
        return sendError(
          res,
          "Username sudah digunakan. Silakan pilih yang lain.",
          409
        );
      }

      const usernameCheck = validateUsername(username);
      if (!usernameCheck.isValid) {
        return sendError(res, usernameCheck.message!, 400);
      }

      const passwordCheck = validatePassword(password);
      if (!passwordCheck.isValid) {
        return sendError(res, passwordCheck.message || "Password lemah", 400);
      }

      const passwordHash = await bcrypt.hash(password, 12);

      const newUser = await Users.create({
        username,
        password: passwordHash,
        roleId,
        isActive,
      });

      const responseData = {
        id: newUser.id,
        username: newUser.username,
        roleId: newUser.roleId,
        isActive: newUser.isActive,
        createdAt: newUser.createdAt,
      };

      return sendSuccess(res, responseData, "Pengguna berhasil dibuat.", 201);
    } catch (error) {
      return sendError(res, "Gagal membuat pengguna", 500, error);
    }
  }
```

Penjelasan Singkat Kode (Fokus Utama: Bcrypt)

Fungsi ini menangani pendaftaran pengguna baru dengan prioritas utama pada keamanan kata sandi.

A. Validasi dan Pencegahan Konflik

- **Pengecekan *Required Fields*:** Memastikan input dasar (username, password, roleId) telah terisi.
- **Pengecekan Keunikan *Username***: Mencari *username* di *database*. Jika ditemukan, mengembalikan status **409 Conflict** (sudah digunakan).
- **Validasi Kualitas Kata Sandi:** Menggunakan validatePassword(password) untuk memastikan kata sandi pengguna memenuhi standar keamanan minimum (misalnya, panjang dan kompleksitas).

B. Inti Keamanan: Hashing Bcrypt

Ini adalah langkah paling penting untuk melindungi data sensitif pengguna.

- **Baris Kunci:** const passwordHash = await bcrypt.hash(password, 12);
- **Mekanisme:**
    - Fungsi bcrypt.hash() mengambil kata sandi mentah (password) dan mengubahnya menjadi hash yang aman dan tidak dapat di-dekripsi (diubah kembali ke teks asli).
    - Parameter 12 adalah Cost Factor (jumlah *salt rounds*), yang menentukan tingkat kesulitan proses *hashing*. Nilai 12 dipilih untuk memberikan perlindungan kuat terhadap serangan *brute force* karena membutuhkan waktu komputasi yang signifikan.
    - Hasil passwordHash (yang mencakup *salt* unik) adalah satu-satunya data yang disimpan di *database* (newUser.create). Ini memastikan bahwa kata sandi asli tidak pernah tersimpan, melindungi pengguna dari kebocoran data.

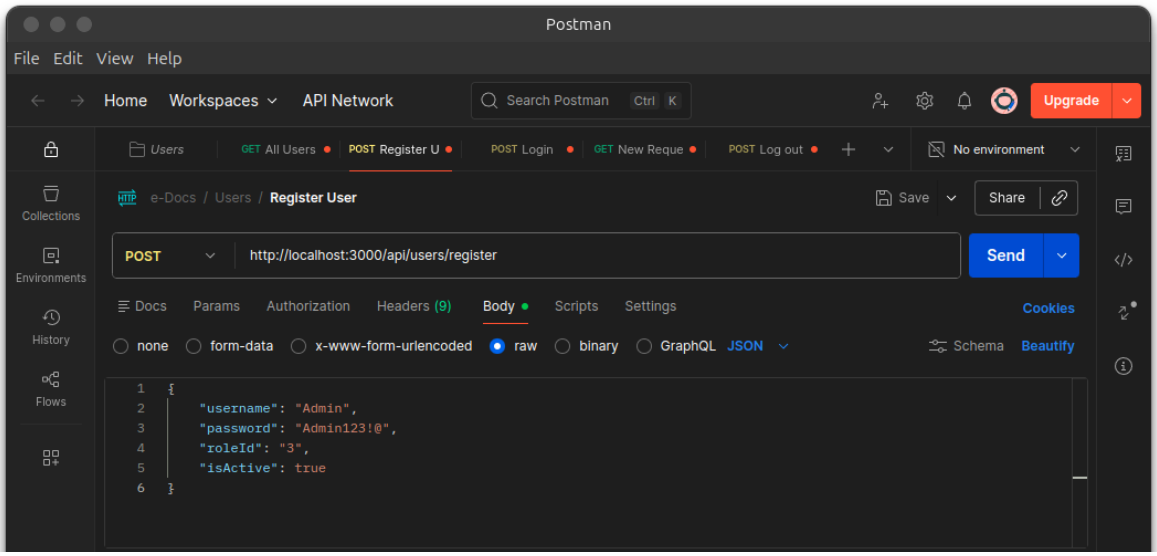
C. Respons

- Setelah berhasil, sistem mengembalikan status **201 Created** dan hanya menampilkan data aman (seperti ID dan *username*), tidak pernah menyertakan kata sandi atau *hash* ke klien.

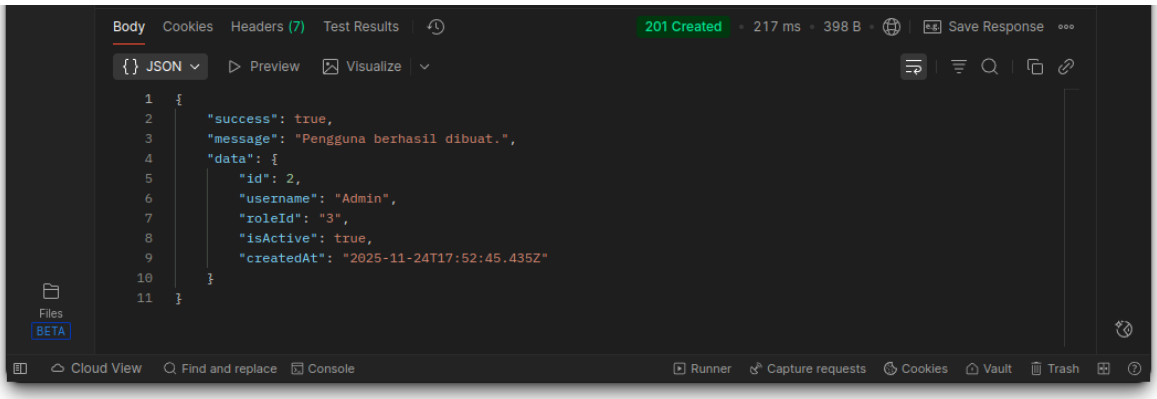
Contoh Penggunaan di Postman dan Respons

Tujuan: Kode ini memproses permintaan POST ke *endpoint* registrasi (misalnya, /api/users/register).

Body Request (JSON): Anda akan mengirim data *username*, *password*, dan *roleId*.



Contoh Respons Sukses (Status 201 Created):



2. Kode Implementasi: Login (postLogin)

Ini adalah fungsi di *Controller* yang menangani proses otentikasi.

```
public async postLogin(req: Request, res: Response): Promise<Response> {
    try {
      const { username, password } = req.body as LoginBody;

      const user = await Users.findOne({ where: { username } });
      if (!user) {
        return sendError(res, "Username atau Password salah", 401);
      }

      if (!user.isActive) {
        return sendError(
          res,
          "Akun Anda dinonaktifkan. Silakan hubungi Admin.",
          403
        );
      }

      const isMatch = await bcrypt.compare(password, user.password);
      if (!isMatch) {
        return sendError(res, "Username atau Password salah", 401);
      }

      const payload = {
        id: user.id,
        roleId: user.roleId,
        username: user.username,
      };

      const token = generateToken(payload);

      return sendSuccess(
        res,
        {
          token: token,
          user: {
            id: user.id,
            username: user.username,
            roleId: user.roleId,
          },
        },
        "Login Berhasil"
      );
    } catch (error) {
      return sendError(res, "Gagal Login", 500, error);
    }
  }
```

Penjelasan Singkat:

Fungsi ini melakukan tiga langkah keamanan utama:

1. **Verifikasi Pengguna:** Mencari *username* di database dan memastikan akun **aktif** (isActive).
2. **Verifikasi Kata Sandi (Bcrypt):** Baris await bcrypt.compare(password, user.password) membandingkan kata sandi yang dimasukkan dengan *hash* yang tersimpan secara aman. **Ini adalah inti keamanan otentikasi.**
3. **Pemberian Akses (JWT):** Jika verifikasi berhasil, fungsi membuat *payload* dan memanggil generateToken(payload) untuk menghasilkan **Token Akses** yang dikirimkan kembali ke pengguna.

Utility Pembuatan JWT (generateToken)

Ini adalah fungsi yang membuat token sesi setelah *login* berhasil.

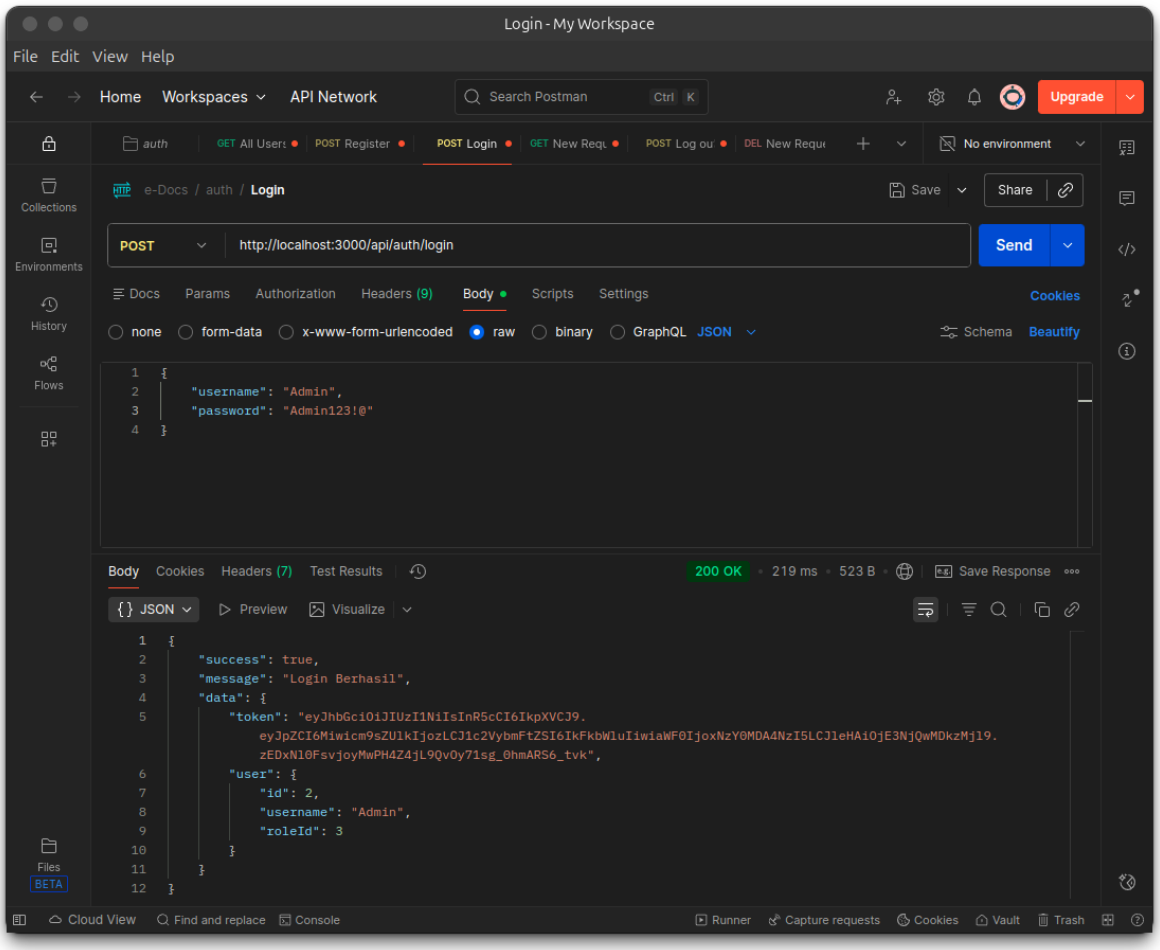
```
import * as jwt from "jsonwebtoken";

interface JWTPayload {
  id: number;
  roleId: number;
  username: string;
}

export function generateToken(payload: JWTPayload): string {
  const SECRET_KEY_STRING = process.env.JWT_SECRET || "rahasia_negara_api";
  const expiresIn = process.env.JWT_EXPIRES_IN || "1d";

  const secretKey = Buffer.from(SECRET_KEY_STRING, "utf8");

  const expiry: string = expiresIn;

  return jwt.sign(payload, secretKey, {
    expiresIn: expiry as jwt.SignOptions["expiresIn"],
  });
}
```

Penjelasan Singkat:

Fungsi ini bertanggung jawab untuk mengubah data pengguna menjadi kunci sesi yang aman.

1. **Kunci Rahasia:** Mengambil SECRET_KEY dari lingkungan (process.env). Kunci ini sangat penting untuk menandatangani (*sign*) token, menjamin keasliannya.
2. **Masa Berlaku:** Mengatur expiresIn (masa berlaku token), mencegah token digunakan selamanya jika dicuri.
3. **Penandatanganan:** Menggunakan jwt.sign() untuk menggabungkan data pengguna (payload) dengan kunci rahasia. Hasilnya adalah JSON Web Token (JWT) yang akan digunakan pengguna untuk mengakses *endpoint* terproteksi.

Contoh Penggunaan di Postman dan Respons



Detail Pengujian:

- **Endpoint:** POST /api/auth/login (atau sesuai konfigurasi Anda)
- **Request Body:** Berisi *username* dan *password* yang dikirimkan dalam format JSON.
- **Status Respons:** 200 OK (Login Berhasil).

- **Data Respons:** Sistem berhasil mengembalikan **token JWT** yang merupakan kunci akses untuk sesi pengguna, beserta data dasar pengguna (id, username, roleId).

3. Implementasi login

A. Fungsi Middleware (authenticateToken)

Fungsi ini dijalankan **sebelum** *controller* diakses, bertugas memverifikasi keaslian Token JWT yang dikirim oleh pengguna.

```ts
import { Request, Response, NextFunction } from "express";
import * as jwt from "jsonwebtoken";
import { sendError } from "../utils/response.utils";

export const authenticateToken = (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  const authHeader = req.headers["authorization"];

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

  if (!token) {
    return sendError(res, "Akses ditolak. Token tidak ditemukan.", 401);
  }

  const SECRET_KEY = process.env.JWT_SECRET || "rahasia_negara_api";

  jwt.verify(token, SECRET_KEY, (err: any, user: any) => {
    if (err) {

      if (err.name === "TokenExpiredError") {
        return sendError(
          res,
          "Sesi Anda telah berakhir. Silakan login ulang.",
          401
        );
      }

      if (err.name === "JsonWebTokenError") {
        return sendError(res, "Token tidak valid. Silakan login ulang.", 401);
      }

      return sendError(res, "Autentikasi gagal.", 401);
    }

    (req as any).user = user;
    next();
  });
};

```

Penjelasan Singkat:

- **Pengecekan Token:** Kode mengambil token dari *header* Authorization (format: Bearer [token]). Jika token tidak ada, akses ditolak (401).
- **Verifikasi JWT:** Fungsi jwt.verify() mencoba memecahkan token menggunakan **Kunci Rahasia Server**.
    - Jika verifikasi gagal (misalnya, token kedaluwarsa atau dimanipulasi), akses ditolak (401).
    - Jika berhasil, *payload* token (informasi pengguna: id, roleId, dll.) dilampirkan ke objek *request* ((req as any).user = user;) dan fungsi memanggil next(), mengizinkan akses ke *controller*.

B. Router (users.route.ts)

Kode ini mendefinisikan *endpoint* dan menerapkan *middleware* proteksi.

```
import { Router } from "express";
import UserController from "../controllers/users.controller";
import { authenticateToken } from "../middleware/auth.middleware";

const router = Router();

router.get(
  "/",
  authenticateToken,
  UserController.getAllUsers.bind(UserController)
);

export default router;
```

Penjelasan Singkat:

- **Proteksi Rute:** Perhatikan *route* router.get("/"). *Middleware* authenticateToken diletakkan di antara *path* dan *controller*.
- **Fungsi:** Ini memastikan bahwa siapa pun yang mencoba mengakses /api/users (untuk memanggil UserController.getAllUsers) WAJIB memiliki Token JWT yang valid dan belum kedaluwarsa. Tanpa token yang valid, permintaan akan dihentikan oleh *middleware* pada Kode 1.

C. Controller (getAllUsers)

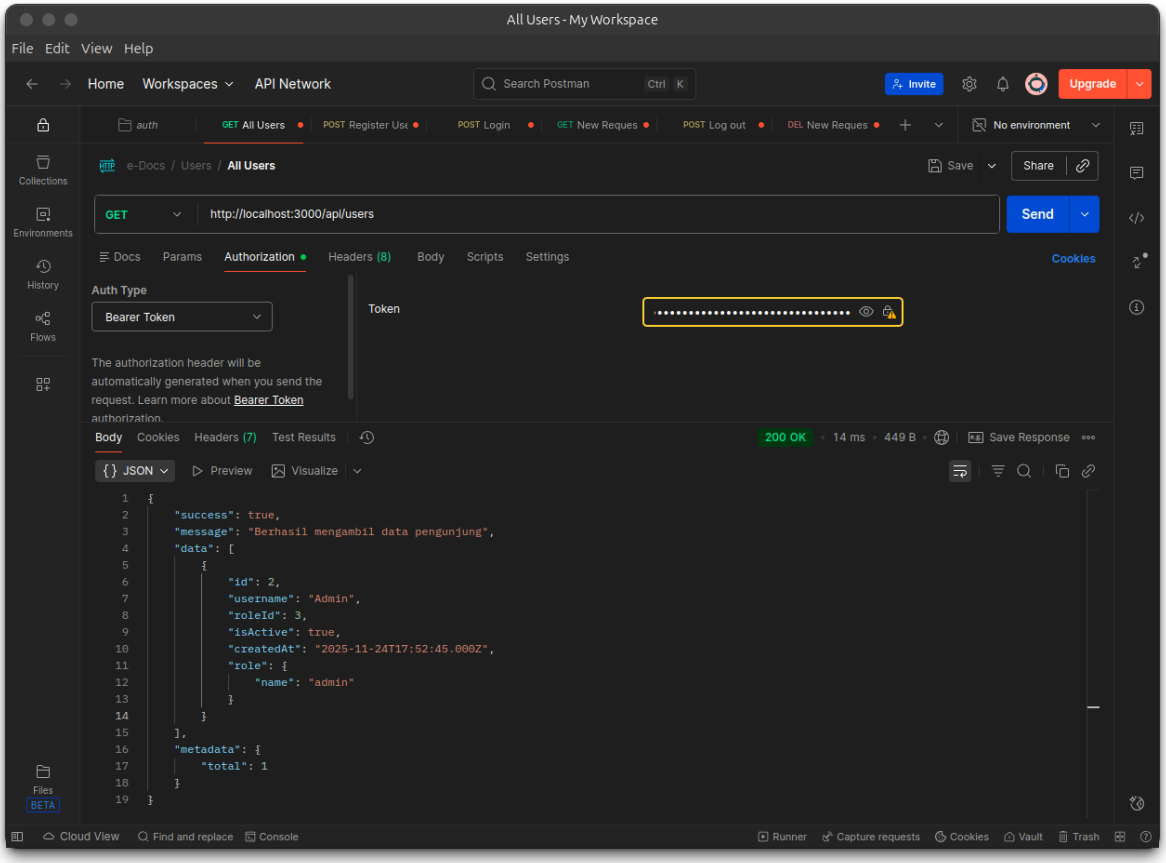
Fungsi ini dijalankan **HANYA JIKA** *middleware* pada Kode 1 berhasil diverifikasi.

```
public async getAllUsers(req: Request, res: Response): Promise<Response> {
    try {
      const users = await Users.findAll({
        include: [
          {
            model: Roles,
            as: "role",
            attributes: ["name"],
          },
        ],
        attributes: { exclude: ["password", "updatedAt"] },
      });

      return sendSuccess(
        res,
        users,
        "Berhasil mengambil data pengunjung",
        200,
        {
          total: users.length,
        }
      );
    } catch (error) {
      return sendError(res, "Gagal Mengambil data pengguna", 500, error);
    }
  }
```

Penjelasan Singkat:

- **Fungsi:** Bertugas mengambil semua data pengguna (Users.findAll) dan menyertakan data *role* terkait.
- **Keamanan Data:** Penggunaan attributes: { exclude: ["password", "updatedAt"] } memastikan bahwa kata sandi pengguna tidak pernah dikirim dalam respons API, bahkan kepada pengguna yang sudah terotentikasi.
- **Akses Terjamin:** Karena *route* ini dilindungi oleh authenticateToken, kita yakin bahwa *request* yang mencapai fungsi ini adalah permintaan yang sah.

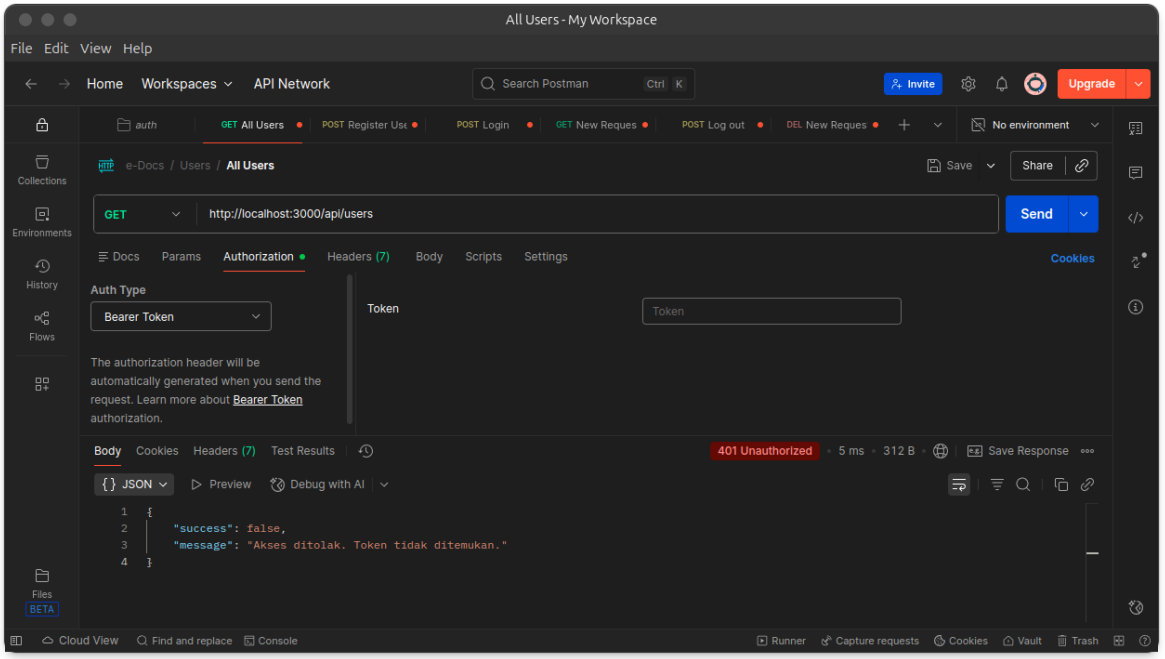
4. Contoh Penggunaan di Postman: Akses dengan Token JWT



Detail Pengujian:

- **Endpoint:** GET /api/users
- **Header** Authorization**:** Token JWT yang diperoleh setelah *login* berhasil disisipkan di sini dengan format Bearer [token_jwt_anda].
- **Status Respons:** 200 OK.
- **Data Respons:** Berhasil mengambil daftar pengguna, sesuai dengan fungsionalitas getAllUsers.

5. Contoh Penggunaan di Postman: Akses Tanpa Token JWT



Detail Pengujian:

- **Endpoint:** GET /api/users
- **Header** Authorization**:** Tidak disertakan atau disertakan dengan token yang tidak valid/kedaluwarsa.
- **Status Respons:** 401 Unauthorized.
- **Data Respons:** Sistem mengembalikan pesan error dari *middleware* authenticateToken (misalnya, "Akses ditolak. Token tidak ditemukan." atau "Token tidak valid.").

6. Mekanisme Proteksi Token JWT

Mekanisme ini bekerja dengan memanfaatkan tiga komponen utama yang telah kita bahas sebelumnya:

- **Token JWT dari Login:** Setelah pengguna berhasil *login*, mereka menerima sebuah **Token JWT**. Ini adalah bukti identitas mereka.
- **authenticateToken Middleware:** Ini adalah penjaga gerbang. Sebelum setiap permintaan ke *endpoint* yang dilindungi (seperti GET /api/users), *middleware* ini akan:
    1. Mengekstrak Token JWT dari *header* Authorization (*request*).
    2. Memverifikasi Token JWT tersebut menggunakan kunci rahasia server.

3. Jika token valid dan belum kedaluwarsa, permintaan diizinkan untuk melanjutkan ke *controller*.
4. Jika token tidak ada, tidak valid, atau kedaluwarsa, *middleware* akan segera menghentikan permintaan dan mengirimkan respons error (401 Unauthorized atau 403 Forbidden).

- **Controller yang Aman (getAllUsers):** Fungsi *controller* getAllUsers hanya akan dieksekusi jika token sudah berhasil diverifikasi oleh *middleware*. Ini menjamin bahwa data pengguna hanya dapat diakses oleh pengguna yang sudah terotentikasi.