LAPORAN
DOKUMENTASI

By Muhammad Rizki Alamsyah

Back-End

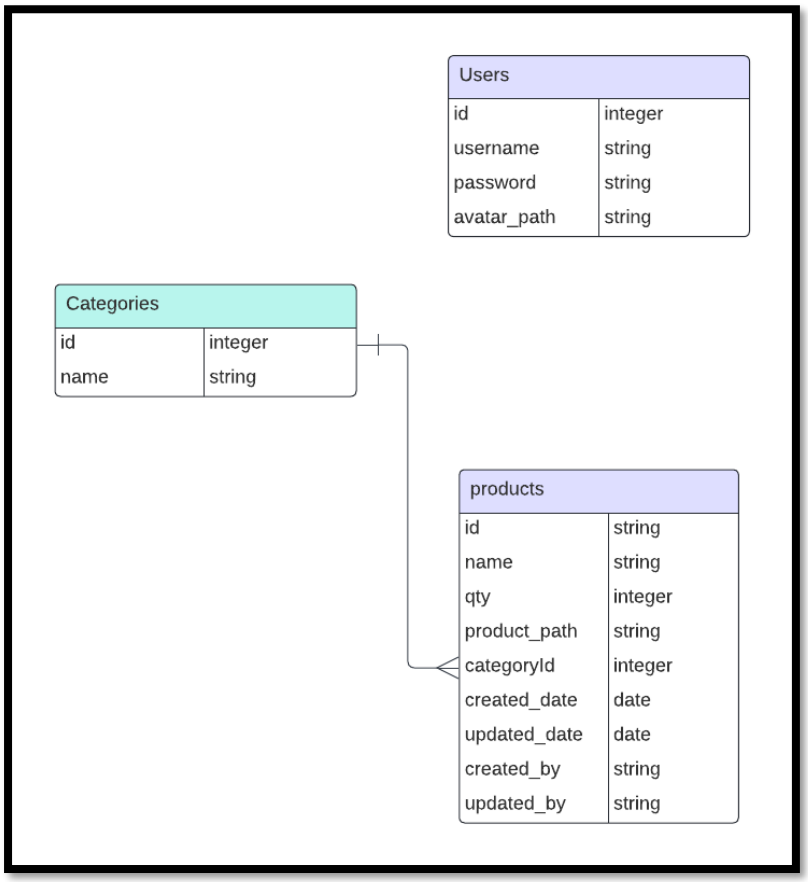
Tools

1. Visual Studio Code
2. Postman
3. Ngrok

Library

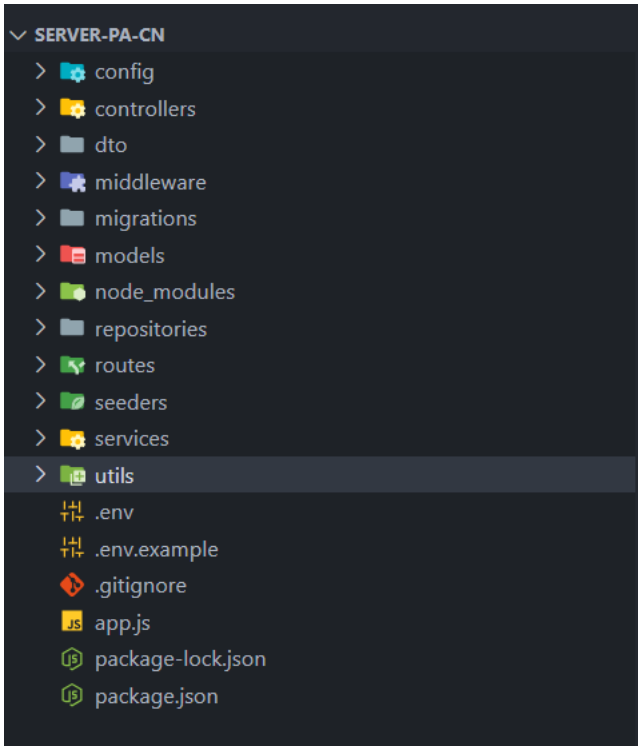
1. bcrypt
2. cloudinary
3. dotenv
4. express
5. jsonwebtoken
6. multer
7. pg
8. sequelize
9. nodemon
10. sequelize-cli

Arsitektur Table



Arsitektur Folder

Arsitektur folder pada project saya ini menggunakan pola MVC (Model-View-Controller) dengan penambahan lapisan Repository dan Service membantu dalam memisahkan tanggung jawab dalam aplikasi. Pola ini memudahkan pengelolaan kode, meningkatkan keterbacaan, dan mempermudah pengujian. Berikut adalah penjelasan tentang arsitektur folder ini :



Penjelasan Tiap Bagian

1. Controllers

   Folder controllers berisi file kontroler yang menangani logika aplikasi dan mengarahkan aliran data antara model dan view. Kontroler menerima input dari klien, memprosesnya (dengan bantuan layanan), dan mengirimkan respons kembali ke klien.

2. Services

   Folder services berisi logika bisnis aplikasi. Layanan ini memanggil metode yang ada di repository dan menggabungkan data yang dibutuhkan untuk memenuhi permintaan dari kontroler.

3. Repositories

   Folder repositories berisi kode yang berinteraksi langsung dengan basis data. Repository ini memastikan bahwa logika akses data dipisahkan dari logika bisnis dan kontroler.

4. Models

   Folder models berisi definisi struktur data dan relasi antar tabel dalam basis data menggunakan ORM (Object-Relational Mapping) seperti Sequelize.

5. Routes

   Folder routes berisi definisi rute yang menghubungkan permintaan HTTP dengan kontroler yang tepat.

6. Middleware

Folder middleware berisi kode yang dijalankan di antara permintaan HTTP dan respons, seperti otentikasi atau pengunggahan file.

7. Utils

   Folder utils berisi utilitas umum yang digunakan di seluruh aplikasi, seperti fungsi hash kata sandi, utilitas JWT, dan helper respons.

8. Migrations

   Folder migrations berisi file migrasi yang digunakan untuk membuat atau memodifikasi tabel dalam basis data.

9. Seeders

   Folder seeders berisi file seeder yang digunakan untuk mengisi basis data dengan data awal

Penjelasan Code

App.js

```
require('dotenv').config();
const express = require('express');
const app = express();
const port = process.env.PORT || 9000;

const routes = require('./routes');
const errorHandler = require('./middleware/errorHandler');

app.use(express.json());
app.use(express.urlencoded({ extended: true }));

app.use(routes);

app.use(errorHandler);

app.listen(port, () => {
  console.log(`App is listening on ${port}`);
});
```

Kode di atas merupakan aplikasi server dasar menggunakan Express.js. Pertama, kode mengimpor konfigurasi dari file .env menggunakan dotenv. Kemudian, modul express diimpor dan aplikasi Express baru dibuat. Port server ditentukan dari variabel lingkungan PORT, atau default ke 9000 jika variabel tersebut tidak ada. Aplikasi kemudian mengimpor dan menggunakan rute yang didefinisikan dalam folder routes serta middleware untuk menangani kesalahan (errorHandler). Middleware express.json() dan express.urlencoded({ extended: true }) digunakan untuk menguraikan permintaan dengan format JSON dan URL-encoded. Akhirnya, server dijalankan dan mendengarkan pada port yang ditentukan, dengan pesan log yang mencatat bahwa server sedang berjalan pada port tersebut.

Middleware

1. **auth.js**

```
const jwt = require('jsonwebtoken');
```

```
const { handleErrorResponse } = require('../utils/responseHelper');
const jwtSecret = process.env.JWT_SECRET || 'your_jwt_secret';

const authenticateToken = (req, res, next) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1];

  if (token == null) {
    return handleErrorResponse(res, 401, 'Token not provided');
  }

  jwt.verify(token, jwtSecret, (err, user) => {
    if (err) {
      return handleErrorResponse(res, 403, 'Invalid token');
    }
    req.user = user;
    next();
  });
};

module.exports = authenticateToken;
```

Kode di atas merupakan middleware Express untuk mengautentikasi token JWT. Middleware ini pertama-tama mengambil token dari header Authorization dari permintaan HTTP. Jika tidak ada token yang ditemukan, respon akan mengembalikan status 401 dengan pesan 'Token not provided' menggunakan fungsi handleErrorResponse. Jika token ada, middleware menggunakan jwt.verify untuk memverifikasi token tersebut dengan kunci rahasia yang disimpan dalam variabel lingkungan JWT_SECRET atau menggunakan default 'your_jwt_secret'. Jika verifikasi gagal, respon akan mengembalikan status 403 dengan pesan 'Invalid token'. Jika verifikasi berhasil, middleware menambahkan objek user yang terdekripsi ke objek req dan memanggil fungsi next() untuk melanjutkan ke middleware berikutnya. Terakhir, middleware ini diekspor untuk digunakan di tempat lain dalam aplikasi.

2. Upload.js

```
const multer = require('multer');
const cloudinary = require('../config/cloudinary');
```

```
const storage = multer.memoryStorage();

const fileFilter = (req, file, cb) => {
  const allowedMimeTypes = [
    'image/bmp',
    'image/jpeg',
    'image/x-png',
    'image/png',
    'image/gif',
    'image/svg+xml',
    'image/webp',
    'image/apng',
    'image/img',
    'image/imgp',
  ];
  const res = req.res;
  if (allowedMimeTypes.includes(file.mimetype)) {
    cb(null, true);
  } else {
    const error = res.status(400).json({
      status: 400,
      message: 'Hanya diperbolehkan untuk mengunggah file gambar
(JPG, PNG)!',
    });
  }
};

const upload = multer({
  storage: storage,
  fileFilter: fileFilter,
});

const uploadImageToCloudinary = (image) => {
  return new Promise((resolve, reject) => {
    cloudinary.uploader
      .upload_stream(
```

```
        { folder: 'pa-cn/products', resource_type: 'auto' },
        (error, result) => {
          if (result) {
            resolve(result.secure_url);
          } else {
            reject(error);
          }
        }
      )
      .end(image.buffer);
  });
};

const uploadAvatarToCloudinary = (buffer) => {
  return new Promise((resolve, reject) => {
    const stream = cloudinary.uploader.upload_stream(
      { folder: 'pa-cn/avatars', resource_type: 'auto' },
      (error, result) => {
        if (result) {
          console.log('Cloudinary upload result:', result);
          resolve(result.secure_url);
        } else {
          console.error('Cloudinary upload error:', error);
          reject(error);
        }
      }
    );
    stream.end(buffer);
  });
};

module.exports = { upload, uploadImageToCloudinary,
uploadAvatarToCloudinary };
```

Kode di atas mengonfigurasi middleware `multer` untuk menangani unggahan file gambar dan integrasi dengan Cloudinary untuk menyimpan gambar di cloud. `multer` dikonfigurasi dengan penyimpanan dalam memori (`memoryStorage`) dan filter file yang hanya mengizinkan tipe MIME tertentu seperti JPG, PNG, GIF, dan lainnya. Jika file

yang diunggah bukan gambar yang diizinkan, respons akan dikirimkan dengan status 400 dan pesan kesalahan. Fungsi `uploadImageToCloudinary` dan `uploadAvatarToCloudinary` menggunakan `cloudinary.uploader.upload_stream` untuk mengunggah gambar ke folder tertentu di Cloudinary dan mengembalikan URL gambar yang diunggah atau kesalahan jika ada. Kode ini kemudian mengekspor konfigurasi `upload` dari `multer` serta dua fungsi unggah ke Cloudinary untuk digunakan di bagian lain aplikasi.

3. errorHandler.js

```
function errorHandler(err, req, res, next) {
  console.error(err.stack);

  const statusCode = err.status || 500;
  const message = err.message || 'Internal Server Error';

  res.status(statusCode).json({
    error: {
      message: message,
    },
  });
}

module.exports = errorHandler;
```

Kode di atas mendefinisikan middleware errorHandler untuk menangani kesalahan dalam aplikasi Express. Fungsi ini menerima parameter kesalahan (err), permintaan (req), respons (res), dan fungsi berikutnya (next). Saat kesalahan terjadi, tumpukan kesalahan dicetak ke konsol. Middleware ini kemudian menentukan status kode respons, yang diambil dari properti status dari objek kesalahan atau default ke 500 jika tidak ada. Pesan kesalahan juga ditentukan dari properti message dari objek kesalahan atau default ke 'Internal Server Error'. Akhirnya, middleware mengirimkan respons JSON dengan status kode dan pesan kesalahan. Middleware ini diekspor untuk digunakan di bagian lain aplikasi.

Utils

1. bycriptUtil

```
const bcrypt = require('bcrypt');
const saltRounds = 10;

const hashPassword = async (password) => {
  return bcrypt.hash(password, saltRounds);
};

const comparePassword = async (password, hashedPassword) => {
  return bcrypt.compare(password, hashedPassword);
};

module.exports = {
  hashPassword,
  comparePassword,
};
```

Kode di atas menggunakan modul bcrypt untuk menyediakan fungsi hashing dan perbandingan kata sandi. Fungsi hashPassword menerima kata sandi sebagai argumen dan mengembalikan kata sandi yang di-hash menggunakan bcrypt dengan saltRounds yang disetel ke 10, memberikan tingkat kompleksitas tertentu untuk keamanan tambahan. Fungsi comparePassword membandingkan kata sandi biasa dengan kata sandi yang di-hash, mengembalikan nilai boolean yang menunjukkan apakah kedua kata sandi cocok. Kedua fungsi ini diekspor untuk digunakan di bagian lain aplikasi, terutama untuk mengamankan kata sandi pengguna dalam proses pendaftaran dan otentikasi.

2. Jwtutil

```
const jwt = require('jsonwebtoken');
const jwtSecret = process.env.JWT_SECRET || 'your_jwt_secret';

const generateToken = (user) => {
  return jwt.sign({ id: user.id, username: user.username },
jwtSecret, {
    expiresIn: '24h',
  });
};

const verifyToken = (token) => {
```

```
  return new Promise((resolve, reject) => {
    jwt.verify(token, jwtSecret, (err, decoded) => {
      if (err) {
        reject(err);
      } else {
        resolve(decoded);
      }
    });
  });
};

module.exports = { generateToken, verifyToken };
```

Kode di atas menggunakan modul `jsonwebtoken` untuk menghasilkan dan memverifikasi token JWT (JSON Web Token). Fungsi `generateToken` menerima objek pengguna sebagai argumen dan menghasilkan token JWT yang berisi ID dan nama pengguna dengan masa berlaku 24 jam, menggunakan rahasia yang diambil dari variabel lingkungan `JWT_SECRET` atau nilai default 'your_jwt_secret'. Fungsi `verifyToken` menerima token sebagai argumen dan memverifikasi token tersebut menggunakan rahasia yang sama, mengembalikan janji (promise) yang dipecahkan dengan data yang terdekode jika verifikasi berhasil, atau menolak janji dengan kesalahan jika verifikasi gagal. Kedua fungsi ini diekspor untuk digunakan di bagian lain aplikasi, terutama untuk autentikasi dan otorisasi pengguna.

3. responseHelper

```
const wrapResponse = (status, message, data = null) => {
  return {
    status,
    message,
    data,
  };
};

const wrapResponse2 = (status, message) => {
  return {
    status,
    message,
```

```
  };
};

const handleSuccessResponse = (res, statusCode, message, data =
null) => {
  res.status(statusCode).json(wrapResponse(statusCode, message,
data));
};

const handleSuccessResponse2 = (res, statusCode, message) => {
  res.status(statusCode).json(wrapResponse2(statusCode, message));
};

const handleErrorResponse = (res, statusCode, message) => {
  res.status(statusCode).json(wrapResponse2(statusCode, message));
};

module.exports = {
  wrapResponse,
  handleSuccessResponse,
  handleSuccessResponse2,
  handleErrorResponse,
};
```

Kode di atas mengimplementasikan beberapa fungsi utilitas untuk mempermudah penanganan respons HTTP dalam aplikasi Node.js. Fungsi wrapResponse dan wrapResponse2 digunakan untuk membungkus respons dalam format objek standar dengan status, pesan, dan data opsional. wrapResponse menerima status, pesan, dan data, sementara wrapResponse2 hanya menerima status dan pesan.

Selanjutnya, fungsi handleSuccessResponse dan handleSuccessResponse2 digunakan untuk mengirim respons sukses ke klien dengan status kode HTTP yang ditentukan, pesan yang menjelaskan hasil operasi, serta data jika diperlukan. handleSuccessResponse memanfaatkan wrapResponse untuk membungkus respons, sedangkan handleSuccessResponse2 menggunakan wrapResponse2.

Terakhir, handleErrorResponse digunakan untuk menangani respons ketika terjadi kesalahan dengan mengirimkan status kode HTTP yang sesuai, serta pesan yang

menjelaskan kesalahan tersebut. Fungsi-fungsi ini membantu dalam memenuhi standar respons API dan memudahkan pengelolaan kode yang konsisten dalam menanggapi permintaan HTTP dari aplikasi. Semua fungsi ini diekspor untuk digunakan di berbagai bagian aplikasi yang membutuhkan penanganan respons HTTP yang seragam dan terstruktur.

Implementasi API

1. **API Login, Register, dan Current User**
   a. **Model**

```
'use strict';
const { Model } = require('sequelize');

module.exports = (sequelize, DataTypes) => {
  class Users extends Model {
  }

  Users.init(
    {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true,
      },
      username: {
        type: DataTypes.STRING,
        allowNull: false,
        unique: true,
        validate: {
          notEmpty: true,
        },
      },
      password: {
        type: DataTypes.STRING,
        allowNull: false,
        validate: {
          notEmpty: true,
```

```
        },
      },
      avatar_path: {
        type: DataTypes.STRING,
        allowNull: true,
      },
    },
    {
      sequelize,
      modelName: 'Users',
      tableName: 'users',
    }
  );

  return Users;
};
```

Kode di atas mendefinisikan model "Users" menggunakan Sequelize, sebuah ORM untuk Node.js, yang mewakili tabel "users" dalam basis data. Pertama, mode strict JavaScript diaktifkan untuk menangkap kesalahan umum. Kemudian, kelas Model dari Sequelize diimpor. Model Users didefinisikan dengan mewarisi dari Model dan diinisialisasi dengan skema tabel yang mencakup empat atribut: id, username, password, dan avatar_path. Atribut id adalah primary key yang otomatis bertambah, username adalah string yang harus unik dan tidak boleh kosong, password adalah string yang tidak boleh kosong, dan avatar_path adalah string opsional. Model ini dikonfigurasi untuk menggunakan koneksi Sequelize yang diteruskan sebagai parameter dan diberi nama Users dengan nama tabel users. Terakhir, model ini diekspor untuk digunakan di bagian lain dari aplikasi.

b. Migrations

```
'use strict';

module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('users', {
      id: {
        allowNull: false,
```

```
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER,
      },
      username: {
        type: Sequelize.STRING,
        allowNull: false,
        unique: true,
      },
      password: {
        type: Sequelize.STRING,
        allowNull: false,
      },
      avatar_path: {
        type: Sequelize.STRING,
        allowNull: true,
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE,
        defaultValue: Sequelize.literal('CURRENT_TIMESTAMP'),
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE,
        defaultValue: Sequelize.literal('CURRENT_TIMESTAMP'),
      },
    });
  },

  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('users');
  },
};
```

Kode di atas adalah sebuah migrasi untuk membuat dan menghapus tabel "users" menggunakan Sequelize. Dalam bagian `up`, tabel "users" didefinisikan dengan beberapa kolom: `id`, yang merupakan primary key auto increment yang tidak boleh

null; `username`, yang merupakan string unik dan tidak boleh null; `password`, yang merupakan string tidak boleh null; `avatar_path`, yang merupakan string opsional; serta `createdAt` dan `updatedAt`, yang merupakan tanggal dengan nilai default timestamp saat ini dan tidak boleh null. Bagian `down` mendefinisikan tindakan untuk menghapus tabel "users" jika diperlukan, memastikan bahwa skema basis data dapat dikembalikan ke keadaan sebelum migrasi dijalankan. Kode ini diekspor sebagai modul untuk digunakan dalam manajemen migrasi basis data menggunakan Sequelize.

c. Seeders

```
'use strict';
const { hashPassword } = require('../utils/bcryptUtils');

module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.bulkDelete('users', null, {});
    await queryInterface.sequelize.query(
      'ALTER SEQUENCE users_id_seq RESTART WITH 1'
    );

    const hashedPassword1 = await hashPassword('password123');
    const hashedPassword2 = await hashPassword('password456');

    await queryInterface.bulkInsert(
      'users',
      [
        {
          username: 'admin1',
          password: hashedPassword1,
          avatar_path:
            'https://res.cloudinary.com/dmuuypm2t/image/upload/v17
10400255/img_photo1_clcavj.png',
          createdAt: new Date(),
          updatedAt: new Date(),
        },
        {
          username: 'admin2',
```

```
          password: hashedPassword2,
          avatar_path:
            'https://res.cloudinary.com/dmuuypm2t/image/upload/v17
10400256/img_photo2_s4csws.png',
          createdAt: new Date(),
          updatedAt: new Date(),
        },
      ],
      {}
    );
  },

  down: async (queryInterface, Sequelize) => {
    await queryInterface.bulkDelete('users', null, {});
    await queryInterface.sequelize.query(
      'ALTER SEQUENCE users_id_seq RESTART WITH 1'
    );
  },
};
```

Kode di atas adalah sebuah migrasi untuk memasukkan data awal ke dalam tabel "users" menggunakan Sequelize. Dalam fungsi `up`, pertama-tama semua data dalam tabel "users" dihapus dan urutan primary key `users_id_seq` di-reset agar mulai dari 1. Kemudian, dua password (`password123` dan `password456`) di-hash menggunakan fungsi `hashPassword` yang diimpor dari `bcryptUtils`. Setelah itu, dua pengguna (`admin1` dan `admin2`) dengan password yang sudah di-hash dan avatar path dimasukkan ke dalam tabel "users" bersama dengan timestamp `createdAt` dan `updatedAt`. Fungsi `down` mendefinisikan langkah-langkah untuk menghapus semua data dari tabel "users" dan me-reset urutan primary key kembali, memungkinkan rollback migrasi jika diperlukan. Kode ini diekspor sebagai modul untuk digunakan dalam manajemen migrasi data menggunakan Sequelize.

d. Routes

```
const usersRoute = require('express').Router();
```

```
const UserController = require('../controllers/usersControllers');
const { upload } = require('../middleware/upload');
const authenticateToken = require('../middleware/auth');

usersRoute.post(
  '/register',
  upload.single('avatar'),
  UserController.registerUser
);
usersRoute.post('/login', upload.none(),
UserController.loginUser);

usersRoute.get(
  '/current-user',
  authenticateToken,
  UserController.getCurrentUserProfile
);

module.exports = usersRoute;
```

Kode di atas mendefinisikan rute untuk pengguna (usersRoute) menggunakan Express.js. Pertama, rute usersRoute diimpor dari express.Router(), dan UserController diimpor dari controllers/usersControllers. Middleware upload diimpor dari middleware/upload untuk menangani unggahan file, dan authenticateToken diimpor dari middleware/auth untuk menangani otentikasi token.

Ada tiga rute yang ditentukan:

1. Rute POST /register menggunakan middleware upload.single('avatar') untuk menangani unggahan avatar tunggal dan memanggil UserController.registerUser untuk mendaftarkan pengguna baru.
2. Rute POST /login menggunakan middleware upload.none() untuk memastikan tidak ada file yang diunggah dan memanggil UserController.loginUser untuk proses login pengguna.

3. Rute GET /current-user menggunakan middleware authenticateToken untuk memastikan pengguna terotentikasi sebelum memanggil UserController.getCurrentUserProfile untuk mendapatkan profil pengguna saat ini.

Lalu, usersRoute diekspor sebagai modul untuk digunakan dalam aplikasi utama.

e. Controller

```
const UserService = require('../services/usersService');
const {
  handleSuccessResponse,
  handleSuccessResponse2,
  handleErrorResponse,
} = require('../utils/responseHelper');
const UserDto = require('../dto/userDto');

class UserController {
  static async registerUser(req, res) {
    const { username, password } = req.body;

    try {
      const avatarBuffer = req.file ? req.file.buffer : null;
      console.log('Avatar buffer:', avatarBuffer);
      const newUser = await UserService.registerUser(
        username,
        password,
        avatarBuffer
      );
      handleSuccessResponse2(res, 201, 'User registered
successfully');
    } catch (error) {
      handleErrorResponse(res, 400, error.message);
    }
  }

  static async loginUser(req, res) {
```

```
    const { username, password } = req.body;

    try {
      const { token, user } = await
UserService.loginUser(username, password);
      const userDto = new UserDto(user, token);
      handleSuccessResponse(res, 200, 'User logged in
successfully', userDto);
    } catch (error) {
      handleErrorResponse(res, 400, error.message);
    }
  }

  static async getCurrentUserProfile(req, res) {
    try {
      const userId = req.user.id;
      const userProfile = await
UserService.getCurrentUserProfile(userId);
      handleSuccessResponse(
        res,
        200,
        'User profile retrieved successfully',
        userProfile
      );
    } catch (error) {
      handleErrorResponse(res, 400, error.message);
    }
  }
}

module.exports = UserController;
```

Kode di atas mendefinisikan UserController, sebuah kelas yang menangani berbagai operasi terkait pengguna dalam aplikasi Node.js berbasis Express.js. UserController memanfaatkan UserService untuk melaksanakan logika bisnis dan responseHelper untuk mengelola respons HTTP.

1. **Register User**

Metode registerUser bertugas untuk mendaftarkan pengguna baru. Ia menerima username, password, dan file avatar dari permintaan (req.body dan req.file). File avatar disimpan sebagai buffer. UserService.registerUser dipanggil dengan parameter tersebut untuk membuat pengguna baru dalam sistem. Jika berhasil, fungsi handleSuccessResponse2 mengirim respons sukses dengan status 201 dan pesan "User registered successfully". Jika ada kesalahan, handleErrorResponse mengirim respons kesalahan dengan status 400 dan pesan kesalahan yang relevan.

2. **Login User**

   Metode loginUser bertugas untuk mengotentikasi pengguna. Ia menerima username dan password dari permintaan (req.body). UserService.loginUser dipanggil dengan parameter tersebut untuk memverifikasi kredensial pengguna. Jika otentikasi berhasil, ia mengembalikan token dan data pengguna, yang kemudian dikemas dalam UserDto. handleSuccessResponse mengirim respons sukses dengan status 200 dan data pengguna dalam DTO. Jika ada kesalahan, handleErrorResponse mengirim respons kesalahan dengan status 400 dan pesan kesalahan yang relevan.

3. **Get Current User Profile**

   Metode getCurrentUserProfile bertugas untuk mengambil profil pengguna saat ini. Ia menggunakan ID pengguna yang diperoleh dari token otentikasi (req.user.id). UserService.getCurrentUserProfile dipanggil dengan userId untuk mengambil data profil pengguna. Jika berhasil, handleSuccessResponse mengirim respons sukses dengan status 200 dan data profil pengguna. Jika ada kesalahan, handleErrorResponse mengirim respons kesalahan dengan status 400 dan pesan kesalahan yang relevan.

Akhirnya, UserController diekspor sebagai modul untuk digunakan dalam pengaturan rute aplikasi, memungkinkan pemisahan logika bisnis dan penanganan rute yang bersih dan terstruktur.

f. Repository

```
const { Users } = require('../models');

class UserRepository {
```

```
  static async findByUsername(username) {
    return await Users.findOne({ where: { username } });
  }

  static async create(user) {
    return await Users.create(user);
  }

  static async findById(id) {
    return await Users.findByPk(id);
  }
}

module.exports = { UserRepository };
```

Kode di atas mendefinisikan UserRepository, sebuah kelas yang bertanggung jawab untuk berinteraksi dengan model Users (atau User, tergantung pada implementasi model) dalam basis data menggunakan Sequelize. Ini adalah bagian dari pola desain Repository yang digunakan untuk memisahkan logika basis data dari logika bisnis dalam aplikasi Node.js.

1. **findByUsername(username)**

   Metode findByUsername digunakan untuk mencari pengguna berdasarkan nama pengguna (username). Dalam implementasinya, ia menggunakan Users.findOne dari Sequelize untuk menemukan entri pengguna yang cocok dengan kriteria yang diberikan dalam objek { where: { username } }. Metode ini mengembalikan objek pengguna jika ditemukan, atau null jika tidak ada pengguna dengan username yang diberikan.

2. **create(user)**

   Metode create digunakan untuk membuat entri baru dalam tabel pengguna (Users). Ia menerima objek user yang berisi data untuk dibuat, seperti username, password, dan lainnya. Dengan menggunakan Users.create(user) dari Sequelize, metode ini mengembalikan objek pengguna yang baru dibuat setelah disimpan dalam basis data.

3. **findById(id)**

Metode findById digunakan untuk mencari pengguna berdasarkan ID (id). Dengan menggunakan Users.findByPk(id) dari Sequelize, metode ini mencari dan mengembalikan objek pengguna yang memiliki primary key sesuai dengan id yang diberikan. Jika tidak ada pengguna yang ditemukan dengan ID yang diberikan, metode ini mengembalikan null.

Penggunaan UserRepository memungkinkan UserController (seperti yang dijelaskan sebelumnya) untuk memisahkan logika bisnis dari akses langsung ke basis data. Ini meningkatkan keterbacaan dan memungkinkan untuk pengujian yang lebih baik dengan mengisolasi logika akses data ke dalam kelas terpisah.

g. Service

```
const { UserRepository } =
require('../repositories/userRepository');
const { generateToken, verifyToken } =
require('../utils/jwtUtils');
const { uploadAvatarToCloudinary } =
require('../middleware/upload');
const { hashPassword, comparePassword } =
require('../utils/bcryptUtils');

class UserService {
  static async registerUser(username, password, avatarBuffer =
null) {
    try {
      const existingUser = await
UserRepository.findByUsername(username);
      if (existingUser) {
        throw new Error('Username already exists');
      }

      if (password.length < 8) {
        throw new Error('Password must be at least 8 characters
long');
      }
```

```
      const hashedPassword = await hashPassword(password);

      let avatarPath = null;
      if (avatarBuffer) {
        console.log('Avatar buffer in service:', avatarBuffer);
        avatarPath = await uploadAvatarToCloudinary(avatarBuffer);
        console.log('Avatar path:', avatarPath);
      }

      const newUser = await UserRepository.create({
        username,
        password: hashedPassword,
        avatar_path: avatarPath,
      });

      console.log('New user created:', newUser);

      return newUser;
    } catch (error) {
      throw error;
    }
  }

  static async loginUser(username, password) {
    try {
      const user = await UserRepository.findByUsername(username);
      if (!user) {
        throw new Error('Invalid username or password');
      }

      const isPasswordValid = await comparePassword(password,
user.password);
      if (!isPasswordValid) {
        throw new Error('Invalid username or password');
      }

      const token = generateToken(user);
```

```
      return { token, user };
    } catch (error) {
      throw error;
    }
  }

  static async getCurrentUserProfile(userId) {
    try {
      const user = await UserRepository.findById(userId);
      if (!user) {
        throw new Error('User not found');
      }
      return {
        username: user.username,
        avatarPath: user.avatar_path,
      };
    } catch (error) {
      throw error;
    }
  }
}

module.exports = UserService;
```

Kode di atas mendefinisikan UserService, sebuah kelas yang menyediakan fungsi-fungsi untuk berinteraksi dengan data pengguna dalam aplikasi Node.js menggunakan Sequelize dan berbagai utilitas seperti JWT, bcrypt, dan pengelolaan unggahan avatar ke Cloudinary.

1. **registerUser(username, password, avatarBuffer = null)**

   Metode `registerUser` digunakan untuk mendaftarkan pengguna baru. Pertama, ia memeriksa apakah username yang diberikan sudah ada dalam basis data menggunakan `UserRepository.findByUsername(username)`. Jika sudah ada, metode melemparkan error dengan pesan "Username already exists". Selanjutnya, metode memeriksa panjang password; jika kurang dari 8

karakter, ia melemparkan error dengan pesan "Password must be at least 8 characters long". Selanjutnya, password yang diberikan di-hash menggunakan `hashPassword` dari `bcryptUtils`. Jika ada `avatarBuffer`, misalnya dari unggahan avatar dalam permintaan, metode mengunggahnya ke Cloudinary menggunakan `uploadAvatarToCloudinary`. Terakhir, metode membuat entri pengguna baru dalam basis data menggunakan `UserRepository.create` dengan username, password yang sudah di-hash, dan path avatar yang sudah diunggah. Jika berhasil, metode mengembalikan objek pengguna yang baru dibuat.

2. **loginUser(username, password)**

   Metode `loginUser` digunakan untuk mengotentikasi pengguna berdasarkan username dan password yang diberikan. Pertama, metode mencari pengguna dalam basis data menggunakan `UserRepository.findByUsername(username)`. Jika tidak ditemukan pengguna dengan username tersebut, metode melemparkan error dengan pesan "Invalid username or password". Selanjutnya, metode memeriksa kecocokan password yang diberikan dengan password yang tersimpan dalam basis data menggunakan `comparePassword` dari `bcryptUtils`. Jika password tidak cocok, metode juga melemparkan error dengan pesan "Invalid username or password". Jika autentikasi berhasil, metode menghasilkan token JWT menggunakan `generateToken` dan mengembalikan token beserta data pengguna dalam objek.

3. **getCurrentUserProfile(userId)**

   Metode `getCurrentUserProfile` digunakan untuk mengambil profil pengguna berdasarkan ID pengguna (`userId`). Metode mencari pengguna dalam basis data menggunakan `UserRepository.findById(userId)`. Jika tidak ditemukan pengguna dengan ID tersebut, metode melemparkan error dengan pesan "User not found". Jika pengguna ditemukan, metode mengembalikan objek dengan username pengguna dan path avatar dari hasil pencarian.

   Dengan menggunakan `UserService`, logika bisnis terkait pengguna terisolasi dengan baik dari logika akses data yang diimplementasikan dalam `UserRepository`. Setiap metode dalam `UserService` ditujukan untuk endpoint atau operasi yang berbeda dalam aplikasi, memisahkan tanggung

jawab fungsional yang jelas dan memfasilitasi pengujian yang lebih baik serta pemeliharaan kode yang lebih mudah.

h. DTO

```
class UserDto {
  constructor(user, token) {
    this.token = token;
    this.username = user.username;
    this.avatar = user.avatar_path;
  }
}

module.exports = UserDto;
```

Kode di atas mendefinisikan kelas UserDto, yang bertindak sebagai objek untuk menyampaikan data pengguna yang diminta dalam format yang sesuai untuk respons HTTP atau keperluan antarmuka pengguna. Konstruktor kelas ini menerima dua parameter: user dan token. Dari parameter user, kelas ini mengekstrak nilai username dan avatar_path (diubah menjadi avatar untuk tujuan konsistensi atau penyederhanaan) untuk menetapkan nilai properti dalam objek UserDto. Properti token langsung diambil dari parameter token yang diberikan.

Penggunaan kelas UserDto ini berguna saat mengirimkan data pengguna yang terotentikasi sebagai respons dari operasi login atau operasi lain yang memerlukan informasi pengguna. Dengan demikian, kelas ini membantu dalam standarisasi format data yang dikirimkan ke frontend atau aplikasi klien lainnya, dengan hanya menyertakan informasi yang relevan seperti username, avatar, dan token.

Dengan mengimplementasikan UserDto, aplikasi dapat memisahkan antara representasi data internal yang mungkin lebih kompleks (seperti struktur data dalam model pengguna) dengan representasi yang lebih sederhana dan fokus pada kebutuhan pengguna akhir atau respons HTTP yang sesuai. Ini juga meningkatkan keterbacaan dan kemudahan dalam pengelolaan data yang dikirimkan antara berbagai komponen aplikasi.

2. **API Category**
   a. **Model**

```
'use strict';
const { Model } = require('sequelize');

module.exports = (sequelize, DataTypes) => {
  class Categories extends Model {}

  Categories.init(
    {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true,
      },
      name: {
        type: DataTypes.STRING,
        allowNull: false,
      },
    },
    {
      sequelize,
      modelName: 'Categories',
      tableName: 'categories',
      timestamps: false,
    }
  );

  return Categories;
};
```

Kode di atas mengilustrasikan definisi model Categories menggunakan Sequelize dalam sebuah aplikasi Node.js. Model ini berfungsi untuk merepresentasikan dan mengelola data kategori dalam basis data relasional. Dengan menggunakan Model dari Sequelize, kode tersebut mengaktifkan penggunaan ORM yang memungkinkan interaksi mudah dengan tabel categories.

Pada definisi model, kolom-kolom utama yang didefinisikan adalah id dan name. Kolom id diatur sebagai primary key dengan tipe data INTEGER yang auto increment, sehingga nilai id akan secara otomatis bertambah setiap kali sebuah entri baru ditambahkan ke dalam tabel. Kolom name adalah sebuah STRING yang tidak boleh null, menampung nama dari setiap kategori yang disimpan dalam basis data.

Selain itu, konfigurasi tambahan seperti sequelize, modelName, dan tableName diberikan saat inisialisasi model. sequelize adalah objek yang mewakili koneksi ke basis data yang digunakan oleh Sequelize. modelName menetapkan nama model ini dalam Sequelize, sedangkan tableName menentukan nama tabel di basis data yang sesuai dengan model.

b. Migrations

```
'use strict';

module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('categories', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER,
      },
      name: {
        type: Sequelize.STRING,
        allowNull: false,
      },
    });
  },

  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('categories');
  },
};
```

Kode di atas merupakan sebuah file migrasi menggunakan Sequelize, sebuah ORM (Object-Relational Mapping) yang populer untuk Node.js, yang bertujuan untuk mengelola struktur basis data dengan mudah dan terstruktur. Dalam Sequelize, migrasi digunakan untuk mengelola versi skema basis data, yang mencakup pembuatan dan penghapusan tabel serta modifikasi lainnya.

Pada bagian up, fungsi up dijalankan ketika migrasi ini diterapkan ke basis data. Langkah pertama dalam up adalah menggunakan queryInterface.createTable untuk membuat tabel baru dengan nama 'categories'. Tabel ini memiliki dua kolom utama:

1. id: Sebagai primary key dengan tipe data INTEGER yang diatur untuk auto increment, sehingga nilai id akan bertambah otomatis dengan setiap entri baru yang ditambahkan.
2. name: Sebagai kolom STRING yang tidak boleh null (allowNull: false), yang akan menampung nama kategori yang disimpan dalam tabel.

Pada bagian down, fungsi down dijalankan ketika migrasi ini di-rollback atau dibatalkan. Fungsi ini menggunakan queryInterface.dropTable untuk menghapus tabel 'categories' dari basis data, mengembalikan basis data ke kondisi sebelum migrasi ini diterapkan.

c. Seeders

```
'use strict';

module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.bulkDelete('products', null, {});
    await queryInterface.sequelize.query(
      'ALTER SEQUENCE products_id_seq RESTART WITH 1'
    );

    await queryInterface.bulkDelete('categories', null, {});
    await queryInterface.sequelize.query(
      'ALTER SEQUENCE categories_id_seq RESTART WITH 1'
    );
```

```
    await queryInterface.bulkInsert(
      'categories',
      [
        {
          name: 'Electronics',
        },
        {
          name: 'Books',
        },
        {
          name: 'Clothing',
        },
        {
          name: 'Sports',
        },
        {
          name: 'Home & Kitchen',
        },
        {
          name: 'Beauty & Personal Care',
        },
        {
          name: 'Toys & Games',
        },
        {
          name: 'Automotive',
        },
        {
          name: 'Health & Household',
        },
      ],
      {}
    );
  },

  down: async (queryInterface, Sequelize) => {
    await queryInterface.bulkDelete('categories', null, {});
```

```
    await queryInterface.sequelize.query(
      'ALTER SEQUENCE categories_id_seq RESTART WITH 1'
    );
  },
};
```

Kode di atas adalah sebuah file migrasi Sequelize yang bertujuan untuk menginisialisasi dan mengelola data kategori dalam tabel 'categories' di dalam basis data aplikasi. Migrasi ini terdiri dari dua fungsi utama: up dan down, yang masing-masing bertanggung jawab untuk menambahkan dan menghapus data kategori.

Pada fungsi up, langkah-langkah yang dilakukan adalah sebagai berikut:

1. Menghapus semua data yang ada dari tabel 'products' menggunakan queryInterface.bulkDelete. Ini dilakukan untuk membersihkan data produk sebelum melakukan operasi berikutnya.
2. Mengatur ulang nilai dari sequence 'products_id_seq' dengan queryInterface.sequelize.query. Sequence digunakan untuk mengatur incrementing value dari kolom id pada tabel 'products'.
3. Menghapus semua data yang ada dari tabel 'categories' menggunakan queryInterface.bulkDelete. Ini dilakukan untuk membersihkan data kategori sebelum memasukkan kategori baru.
4. Mengatur ulang nilai dari sequence 'categories_id_seq' dengan queryInterface.sequelize.query. Ini dilakukan untuk mengatur ulang incrementing value dari kolom id pada tabel 'categories' setelah menghapus semua data.
5. Menyisipkan kembali data kategori yang baru ke dalam tabel 'categories' menggunakan queryInterface.bulkInsert. Data ini berisi daftar nama-nama kategori seperti 'Electronics', 'Books', 'Clothing', dan lain-lain. Setiap kategori dimasukkan sebagai objek dalam array yang disertakan sebagai parameter pertama dari bulkInsert.

Pada fungsi down, langkah-langkah yang dilakukan adalah membalikkan perubahan yang dilakukan oleh fungsi up:

1. Menghapus semua data yang ada dari tabel 'categories' menggunakan queryInterface.bulkDelete.
2. Mengatur ulang nilai dari sequence 'categories_id_seq' dengan queryInterface.sequelize.query.

d. Routes

```
const categoriesRoute = require('express').Router();
const { CategoriesController } = require('../controllers');
const authenticateToken = require('../middleware/auth');

categoriesRoute.get(
  '/',
  authenticateToken,
  CategoriesController.getAllCategories
);

module.exports = categoriesRoute;
```

Kode di atas mengimplementasikan sebuah router dalam framework Express untuk menangani endpoint terkait kategori dalam aplikasi Node.js. Router tersebut disebut categoriesRoute dan didesain untuk menanggapi permintaan HTTP GET pada path '/'. Endpoint ini memanfaatkan middleware authenticateToken untuk memverifikasi keautentikan pengguna melalui token yang disertakan dalam header permintaan. Setelah verifikasi, permintaan akan diproses oleh CategoriesController.getAllCategories, yang bertanggung jawab untuk mengambil dan mengembalikan semua kategori yang tersedia dalam sistem. Dengan konfigurasi ini, aplikasi dapat mengatur akses ke data kategori dengan aman, memastikan bahwa hanya pengguna yang terautentikasi yang dapat mengakses informasi tersebut.

e. Controller

```
const categoryService =
require('../services/categoryService');
const {
  handleSuccessResponse,
  handleErrorResponse,
} = require('../utils/responseHelper');
```

```
exports.getAllCategories = async (req, res) => {
  try {
    const categories = await
categoryService.getAllCategories();
    handleSuccessResponse(
      res,
      200,
      'Categories fetched successfully',
      categories
    );
  } catch (error) {
    handleErrorResponse(res, 500, error.message);
  }
};
```

Kode di atas adalah bagian dari kontroler dalam sebuah aplikasi Node.js yang menggunakan framework Express. Kontroler ini bertanggung jawab untuk menangani permintaan HTTP GET pada endpoint yang mengambil semua kategori dari sistem. Saat permintaan diterima, fungsi getAllCategories dieksekusi.

Pertama, kontroler ini mengimpor categoryService dari modul ../services/categoryService, yang merupakan bagian dari arsitektur aplikasi yang mengelola logika bisnis terkait kategori. Fungsi getAllCategories dipanggil dari categoryService untuk mengambil semua kategori dari basis data atau sumber data lainnya. Proses ini dilakukan secara asynchronous dengan menggunakan async/await untuk menangani operasi yang mungkin membutuhkan waktu.

Ketika kategori berhasil diambil, respons sukses diproses menggunakan fungsi handleSuccessResponse yang diimpor dari ../utils/responseHelper. Respons ini akan memiliki status HTTP 200 OK dan akan menyertakan pesan sukses "Categories fetched successfully" bersama dengan data kategori yang diperoleh dari categoryService.

Namun, jika terjadi kesalahan selama pengambilan kategori, penanganan kesalahan dilakukan dengan memanggil fungsi handleErrorResponse dari responseHelper. Kesalahan ini akan menghasilkan respons dengan status HTTP 500 Internal Server Error dan akan menyertakan pesan kesalahan yang diambil dari objek error yang dilempar.

f. Repository

```
const { Categories } = require('../models');

class CategoryRepository {
  async findAll() {
    return await Categories.findAll();
  }
}

module.exports = new CategoryRepository();
```

Kode di atas merupakan implementasi dari sebuah repository dalam aplikasi Node.js yang menggunakan Sequelize sebagai ORM (Object-Relational Mapping). Repository ini, yang disebut CategoryRepository, bertanggung jawab untuk mengakses dan mengelola data kategori dari basis data.

Pertama, model Categories diimpor dari modul ../models, yang mengacu pada definisi model kategori yang telah didefinisikan menggunakan Sequelize. Model ini mewakili struktur dan skema dari tabel 'categories' di dalam basis data.

Kelas CategoryRepository memiliki satu metode yaitu findAll, yang dirancang untuk mengambil semua data kategori dari tabel 'categories'. Metode ini diimplementasikan menggunakan async/await untuk memastikan proses query ke basis data berjalan secara asynchronous. Dalam hal ini, findAll menggunakan metode Categories.findAll() dari Sequelize untuk melakukan query dan mengembalikan hasilnya.

Objek CategoryRepository diekspor dari modul menggunakan module.exports = new CategoryRepository(), yang berarti hanya satu instance dari CategoryRepository akan dibuat dan digunakan di seluruh aplikasi. Hal ini memastikan penggunaan yang efisien dari repository pattern, di mana logika akses data terisolasi dan dapat digunakan secara konsisten di seluruh aplikasi.

g. Service

```
const categoryRepository =
require('../repositories/categoryRepository');

class CategoryService {
```

```
  async getAllCategories() {
    return await categoryRepository.findAll();
  }
}

module.exports = new CategoryService();
```

Kode di atas mengimplementasikan sebuah layanan (service) dalam aplikasi Node.js yang bertujuan untuk menyediakan fungsionalitas terkait dengan entitas kategori. Layanan ini, yang disebut CategoryService, bertanggung jawab untuk mengakses data kategori melalui penggunaan repository pattern.

Pertama, categoryRepository diimpor dari ../repositories/categoryRepository. Repository ini sebelumnya telah didefinisikan dan mengimplementasikan metode findAll untuk mengambil semua data kategori dari basis data.

Kelas CategoryService memiliki satu metode yaitu getAllCategories, yang dirancang untuk mengembalikan seluruh kategori yang ada. Metode ini diimplementasikan menggunakan async/await untuk memastikan operasi query ke basis data berjalan secara asynchronous. Dalam hal ini, getAllCategories menggunakan categoryRepository.findAll() untuk meminta semua kategori dari basis data melalui repository yang sesuai.

Objek CategoryService diekspor dari modul menggunakan module.exports = new CategoryService(), yang berarti hanya satu instance dari CategoryService akan dibuat dan digunakan di seluruh aplikasi. Pendekatan ini membantu dalam menjaga state yang konsisten dan efisien penggunaan sumber daya.

3. **API Product**
   a. **Model**

```
'use strict';
const { Model } = require('sequelize');

module.exports = (sequelize, DataTypes) => {
  class Products extends Model {
    static associate(models) {
      Products.belongsTo(models.Categories, {
```

```
      foreignKey: 'categoryId',
      as: 'category',
    });
  }
}

Products.init(
  {
    id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    name: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    qty: {
      type: DataTypes.INTEGER,
      allowNull: false,
    },
    categoryId: {
      type: DataTypes.INTEGER,
      references: {
        model: 'categories',
        key: 'id',
      },
      allowNull: false,
    },
    imageUrl: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    createdAt: {
      type: DataTypes.DATE,
      defaultValue: DataTypes.NOW,
      allowNull: false,
```

```
      },
      updatedAt: {
        type: DataTypes.DATE,
        defaultValue: DataTypes.NOW,
        allowNull: false,
      },
      createdBy: {
       type: DataTypes.STRING,
        allowNull: false,
      },
      updatedBy: {
        type: DataTypes.STRING,
        allowNull: false,
      },
    },
    {
      sequelize,
      modelName: 'Products',
      tableName: 'products',
      timestamps: true,
    }
  );

  return Products;
};
```

Kode di atas merupakan definisi model Sequelize untuk entitas 'Products' dalam aplikasi Node.js yang menggunakan ORM Sequelize untuk berinteraksi dengan basis data. Model ini menggambarkan struktur dan perilaku entitas produk dalam sistem.

Pertama, model 'Products' diinisialisasi dengan menggunakan class Products yang merupakan turunan dari Model yang diimpor dari Sequelize. Di dalam definisi kelas Products, terdapat sebuah static method associate yang digunakan untuk menentukan asosiasi atau hubungan antara model 'Products' dengan model 'Categories'. Dalam kasus ini, 'Products' memiliki relasi 'belongsTo' dengan 'Categories', yang berarti setiap produk memiliki kategori tertentu. Konfigurasi ini dilakukan dengan menggunakan foreign key 'categoryId' yang mengacu pada kolom 'id' di tabel 'categories'.

Dalam bagian Products.init, definisi kolom-kolom dalam tabel 'products' dijelaskan menggunakan objek konfigurasi:

1. id sebagai primary key dengan tipe data INTEGER yang diincrement secara otomatis.
2. name sebagai nama produk dengan tipe data STRING yang tidak boleh kosong.
3. qty sebagai jumlah produk dengan tipe data INTEGER yang tidak boleh kosong.
4. categoryId sebagai foreign key yang mengacu pada tabel 'categories' dengan tipe data INTEGER.
5. imageUrl sebagai URL gambar produk dengan tipe data STRING yang tidak boleh kosong.
6. createdAt dan updatedAt sebagai kolom timestamp yang secara otomatis diatur nilainya saat penambahan atau pembaruan data.
7. createdBy dan updatedBy sebagai kolom STRING yang merekam siapa yang membuat atau memperbarui produk, yang tidak boleh kosong.

Konfigurasi tambahan termasuk pengaturan sequelize yang mewakili koneksi ke basis data, penamaan model modelName sebagai 'Products', penamaan tabel tableName sebagai 'products', dan pengaturan timestamps menjadi true untuk mengaktifkan penggunaan kolom createdAt dan updatedAt.

b. Migrations

```
'use strict';

module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('products', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER,
      },
      name: {
        type: Sequelize.STRING,
        allowNull: false,
```

```
      },
      qty: {
        type: Sequelize.INTEGER,
        allowNull: false,
      },
      categoryId: {
        type: Sequelize.INTEGER,
        allowNull: false,
        references: {
          model: 'categories',
          key: 'id',
        },
      },
      imageUrl: {
        type: Sequelize.STRING,
        allowNull: false,
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE,
        defaultValue: Sequelize.literal('CURRENT_TIMESTAMP'),
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE,
        defaultValue: Sequelize.literal('CURRENT_TIMESTAMP'),
      },
      createdBy: {
        type: Sequelize.STRING,
        allowNull: false,
      },
      updatedBy: {
        type: Sequelize.STRING,
        allowNull: false,
      },
    });
  },
```

```
  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('products');
  },
};
```

Kode di atas merupakan sebuah file migrasi untuk database dalam aplikasi Node.js yang menggunakan Sequelize sebagai ORM untuk berinteraksi dengan basis data. Migrasi ini bertanggung jawab untuk membuat tabel 'products' di dalam basis data serta mendefinisikan struktur kolom-kolom yang diperlukan.

Dalam metode up, yang merupakan bagian dari Sequelize migration, terjadi proses pembuatan tabel 'products' menggunakan queryInterface.createTable(). Tabel ini memiliki beberapa kolom yang telah didefinisikan:

1. id sebagai primary key dengan tipe data INTEGER yang diincrement secara otomatis.
2. name sebagai nama produk dengan tipe data STRING yang tidak boleh kosong.
3. qty sebagai jumlah produk dengan tipe data INTEGER yang tidak boleh kosong.
4. categoryId sebagai foreign key yang mengacu pada tabel 'categories' dengan tipe data INTEGER.
5. imageUrl sebagai URL gambar produk dengan tipe data STRING yang tidak boleh kosong.
6. createdAt dan updatedAt sebagai kolom timestamp yang secara otomatis diatur nilainya saat penambahan atau pembaruan data.
7. createdBy dan updatedBy sebagai kolom STRING yang merekam siapa yang membuat atau memperbarui produk, yang tidak boleh kosong.

Pada kolom categoryId, terdapat definisi references yang menunjukkan bahwa kolom ini merupakan foreign key yang mengacu pada kolom 'id' di tabel 'categories'. Ini mengatur hubungan antara entitas 'products' dengan 'categories', yang memungkinkan untuk mengaitkan setiap produk dengan kategori tertentu.

Di bagian down, yang merupakan metode untuk rollback atau menghapus migrasi, terdapat perintah queryInterface.dropTable('products') yang digunakan untuk menghapus tabel 'products' dari basis data. Hal ini memungkinkan untuk membatalkan perubahan yang telah dilakukan jika diperlukan.

c. **Seeders**

```
'use strict';

module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.bulkDelete('products', null, {});
    await queryInterface.sequelize.query(
      'ALTER SEQUENCE products_id_seq RESTART WITH 1'
    );

    await queryInterface.bulkInsert(
      'products',
      [
        {
          name: 'Laptop',
          qty: 10,
          categoryId: 1,
          imageUrl:
            'https://res.cloudinary.com/dmuuypm2t/image/upload/v17
10400118/img_car_lhk3me.png',
          createdAt: new Date(),
          updatedAt: new Date(),
          createdBy: 'admin',
          updatedBy: 'admin',
        },
        {
          name: 'Book Title',
          qty: 50,
          categoryId: 2,
          imageUrl:
            'https://res.cloudinary.com/dmuuypm2t/image/upload/v17
10400118/img_car_lhk3me.png',
          createdAt: new Date(),
          updatedAt: new Date(),
          createdBy: 'admin',
          updatedBy: 'admin',
```

```
        },
      ],
      {}
    );
  },

  down: async (queryInterface, Sequelize) => {
    await queryInterface.bulkDelete('products', null, {});
    await queryInterface.sequelize.query(
      'ALTER SEQUENCE products_id_seq RESTART WITH 1'
    );
  },
};
```

Kode di atas adalah sebuah file migrasi dalam aplikasi Node.js yang menggunakan Sequelize sebagai ORM untuk berinteraksi dengan basis data. Migrasi ini bertujuan untuk mengelola data awal atau seed data untuk tabel 'products' di dalam basis data.

Pada metode up, yang merupakan bagian dari Sequelize migration, terjadi beberapa proses:

1. bulkDelete('products', null, {}): Ini menghapus semua data yang ada di tabel 'products', memastikan tabel tersebut kosong sebelum memasukkan data baru.
2. sequelize.query('ALTER SEQUENCE products_id_seq RESTART WITH 1'): Perintah ini mengatur ulang sequence untuk kolom 'id' di tabel 'products', dimulai dari nilai 1. Hal ini berguna untuk mengatur ulang auto increment pada primary key setelah penghapusan data.

Setelah itu, dilakukan bulkInsert untuk memasukkan data baru ke dalam tabel 'products'. Dalam kasus ini, dua produk ditambahkan:

1. Produk pertama adalah 'Laptop' dengan jumlah 10 unit, kategori dengan ID 1 (asumsi kategori 'Electronics'), URL gambar produk, dan catatan waktu pembuatan dan pembaruan yang sama, serta ditandai dibuat dan diperbarui oleh 'admin'.

2. Produk kedua adalah 'Book Title' dengan jumlah 50 unit, kategori dengan ID 2 (asumsi kategori 'Books'), URL gambar produk yang sama, dan catatan waktu yang serupa untuk pembuatan dan pembaruan, juga ditandai oleh 'admin'.

Data ini dimasukkan ke dalam tabel menggunakan Sequelize's bulkInsert untuk efisiensi, dengan menyediakan array objek yang merepresentasikan setiap baris data yang akan dimasukkan.

Di bagian down, yang merupakan metode untuk rollback atau menghapus migrasi, dilakukan kembali bulkDelete untuk menghapus semua data dari tabel 'products', dan sequelize.query untuk mengatur ulang sequence 'products_id_seq' kembali ke nilai awal, yaitu 1.

1. Menghapus semua data yang ada dari tabel 'categories' menggunakan queryInterface.bulkDelete.
2. Mengatur ulang nilai dari sequence 'categories_id_seq' dengan queryInterface.sequelize.query.

d. Routes

```
const productsRoute = require('express').Router();
const { ProductsController } = require('../controllers');
const { upload } = require('../middleware/upload');
const authenticateToken = require('../middleware/auth');

productsRoute.get('/', authenticateToken,
ProductsController.getAllProducts);
productsRoute.get('/:id', authenticateToken,
ProductsController.getProductById);
productsRoute.post(
  '/',
  authenticateToken,
  upload.single('image'),
  ProductsController.createProduct
);
productsRoute.put(
  '/:id',
```

```
  authenticateToken,
  upload.single('image'),
  ProductsController.updateProduct
);
productsRoute.delete(
  '/:id',
  authenticateToken,
  ProductsController.deleteProduct
);

module.exports = productsRoute;
```

Kode di atas mendefinisikan sebuah router untuk produk (productsRoute) dalam aplikasi Express.js. Router ini digunakan untuk menangani berbagai operasi terkait produk, seperti pengambilan semua produk, pengambilan produk berdasarkan ID, pembuatan produk baru, pembaruan produk, dan penghapusan produk. Mari kita bahas secara rinci:

1. **Endpoint untuk Mendapatkan Semua Produk (GET /):** Endpoint ini menggunakan middleware authenticateToken untuk memverifikasi token autentikasi sebelum memanggil ProductsController.getAllProducts. Tujuannya adalah untuk membatasi akses hanya untuk pengguna yang terotentikasi sebelum dapat melihat daftar semua produk yang tersedia di aplikasi.
2. **Endpoint untuk Mendapatkan Produk berdasarkan ID (GET /:id):** Endpoint ini juga menggunakan middleware authenticateToken untuk memastikan hanya pengguna yang terotentikasi yang dapat mengaksesnya. ProductsController.getProductById digunakan untuk menangani permintaan ini dan mengembalikan detail produk berdasarkan ID yang diberikan dalam URL.
3. **Endpoint untuk Membuat Produk Baru (POST /):** Saat mengakses endpoint ini, pengguna harus terlebih dahulu melewati autentikasi token (authenticateToken) dan mengunggah file gambar produk menggunakan middleware upload.single('image'). Setelah itu, ProductsController.createProduct dipanggil untuk memproses data yang diterima dan membuat produk baru dalam basis data.

4. **Endpoint untuk Memperbarui Produk (PUT /:id):** Endpoint ini memungkinkan pengguna untuk memperbarui produk berdasarkan ID yang ditentukan dalam URL (:id). Seperti sebelumnya, autentikasi token (authenticateToken) diperlukan untuk mengaksesnya, dan pengguna dapat mengunggah gambar produk baru menggunakan middleware upload.single('image'). ProductsController.updateProduct akan menangani proses memperbarui data produk berdasarkan data yang diterima.
5. **Endpoint untuk Menghapus Produk (DELETE /:id):** Terakhir, endpoint ini memungkinkan pengguna untuk menghapus produk berdasarkan ID yang ditentukan dalam URL. Autentikasi token (authenticateToken) dibutuhkan untuk mengaksesnya, dan ProductsController.deleteProduct akan dipanggil untuk menghapus produk dari basis data.

Setiap endpoint telah dirancang dengan mempertimbangkan keamanan (melalui autentikasi token), serta fungsionalitas untuk mengelola operasi CRUD (Create, Read, Update, Delete) terhadap entitas produk dalam aplikasi. Pendekatan ini memisahkan logika bisnis (diwakili oleh controller) dari logika routing, sehingga memungkinkan untuk pengelolaan kode yang lebih terstruktur dan mudah dipelihara dalam pengembangan aplikasi berbasis Express.js.

e. Controller

```
const productService = require('../services/productService');
const {
  handleSuccessResponse,
  handleSuccessResponse2,
  handleErrorResponse,
} = require('../utils/responseHelper');

exports.getAllProducts = async (req, res) => {
  try {
    const { categoryId, name } = req.query;
    const products = await
productService.getAllProducts(categoryId, name);
    handleSuccessResponse(res, 200, 'Products fetched
successfully', products);
  } catch (error) {
```

```
    handleErrorResponse(res, 500, error.message);
  }
};

exports.getProductById = async (req, res) => {
  try {
    const product = await
productService.getProductById(req.params.id);
    handleSuccessResponse(res, 200, 'Product fetched
successfully', product);
  } catch (error) {
    if (error.message === 'Product not found') {
      handleErrorResponse(res, 404, error.message);
    } else {
      handleErrorResponse(res, 500, error.message);
    }
  }
};

exports.createProduct = async (req, res) => {
  try {
    const product = await productService.createProduct(
      req.body,
      req.file,
      req.user
    );

    handleSuccessResponse(res, 201, 'Product created
successfully', product);
  } catch (error) {
    handleErrorResponse(res, 500, error.message);
  }
};

exports.updateProduct = async (req, res) => {
  try {
    const product = await productService.updateProduct(
```

```
      req.params.id,
      req.body,
      req.file,
      req.user
    );
    handleSuccessResponse(res, 200, 'Product updated
successfully', product);
  } catch (error) {
    if (error.message === 'Product not found') {
      handleErrorResponse(res, 404, error.message);
    } else {
      handleErrorResponse(res, 500, error.message);
    }
  }
};

exports.deleteProduct = async (req, res) => {
  try {
    await productService.deleteProduct(req.params.id);
    handleSuccessResponse2(res, 200, 'Product deleted
successfully');
  } catch (error) {
    if (error.message === 'Product not found') {
      handleErrorResponse(res, 404, error.message);
    } else {
      handleErrorResponse(res, 500, error.message);
    }
  }
};
```

Kode di atas mendefinisikan beberapa fungsi controller dalam file JavaScript yang menangani berbagai endpoint untuk entitas produk dalam aplikasi berbasis Express.js. Setiap fungsi bertanggung jawab untuk menangani permintaan HTTP tertentu yang berkaitan dengan produk. Mari kita bahas setiap endpoint dalam paragraf terpisah.

1. **Endpoint untuk Mendapatkan Semua Produk (GET /):**

Fungsi getAllProducts menangani permintaan untuk mendapatkan semua produk. Fungsi ini mengambil parameter categoryId dan name dari query string, lalu memanggil metode getAllProducts dari productService dengan parameter tersebut. Jika berhasil, fungsi ini mengirimkan respons sukses dengan kode status 200 dan data produk yang ditemukan. Jika terjadi kesalahan, fungsi mengirimkan respons error dengan kode status 500 dan pesan error.

2. **Endpoint untuk Mendapatkan Produk Berdasarkan ID (GET /:id):**

   Fungsi getProductById menangani permintaan untuk mendapatkan detail produk berdasarkan ID yang diberikan dalam URL. Fungsi ini memanggil metode getProductById dari productService dengan parameter ID produk. Jika produk ditemukan, fungsi mengirimkan respons sukses dengan kode status 200 dan detail produk. Jika produk tidak ditemukan, fungsi mengirimkan respons error dengan kode status 404 dan pesan 'Product not found'. Jika terjadi kesalahan lain, fungsi mengirimkan respons error dengan kode status 500 dan pesan error.

3. **Endpoint untuk Membuat Produk Baru (POST /):**

   Fungsi createProduct menangani permintaan untuk membuat produk baru. Fungsi ini memanggil metode createProduct dari productService dengan data produk dari req.body, file gambar dari req.file, dan pengguna yang membuat produk dari req.user. Jika berhasil, fungsi ini mengirimkan respons sukses dengan kode status 201 dan data produk yang baru dibuat. Jika terjadi kesalahan, fungsi mengirimkan respons error dengan kode status 500 dan pesan error.

4. **Endpoint untuk Memperbarui Produk (PUT /:id):**

   Fungsi updateProduct menangani permintaan untuk memperbarui produk berdasarkan ID yang diberikan dalam URL. Fungsi ini memanggil metode updateProduct dari productService dengan parameter ID produk, data produk dari req.body, file gambar dari req.file, dan pengguna yang memperbarui produk dari req.user. Jika produk diperbarui dengan sukses,

fungsi mengirimkan respons sukses dengan kode status 200 dan data produk yang diperbarui. Jika produk tidak ditemukan, fungsi mengirimkan respons error dengan kode status 404 dan pesan 'Product not found'. Jika terjadi kesalahan lain, fungsi mengirimkan respons error dengan kode status 500 dan pesan error.

5. **Endpoint untuk Menghapus Produk (DELETE /:id):**

Fungsi deleteProduct menangani permintaan untuk menghapus produk berdasarkan ID yang diberikan dalam URL. Fungsi ini memanggil metode deleteProduct dari productService dengan parameter ID produk. Jika produk dihapus dengan sukses, fungsi mengirimkan respons sukses dengan kode status 200 dan pesan 'Product deleted successfully'. Jika produk tidak ditemukan, fungsi mengirimkan respons error dengan kode status 404 dan pesan 'Product not found'. Jika terjadi kesalahan lain, fungsi mengirimkan respons error dengan kode status 500 dan pesan error.

f. Repository

```
const { Products, Categories } = require('../models');

class ProductRepository {
  async findAll(categoryId, name) {
    const query = {
      include: [
        {
          model: Categories,
          as: 'category',
          attributes: ['name'],
        },
      ],
      order: [['createdAt', 'DESC']],
    };

    if (categoryId) {
      query.where = { categoryId };
    }
```

```
  if (name) {
    query.where = { ...query.where, name };
  }

  const products = await Products.findAll(query);

  return products.map((product) => {
    const productData = product.get({ plain: true });
    productData.categoryName = productData.category.name;
    delete productData.category;
    return productData;
  });
}

async findById(id) {
  return await Products.findByPk(id, {
    include: [
      {
        model: Categories,
        as: 'category',
        attributes: ['name'],
      },
    ],
  });
}

async create(data) {
  return await Products.create(data);
}

async update(product, data) {
  return await product.update(data);
}

async delete(product) {
  return await product.destroy();
```

```
  }
}

module.exports = new ProductRepository();
```

Kode di atas mendefinisikan kelas ProductRepository dalam file JavaScript yang berfungsi untuk berinteraksi dengan model Products dan Categories di basis data menggunakan Sequelize. Setiap metode dalam kelas ini bertanggung jawab untuk melakukan operasi CRUD (Create, Read, Update, Delete) pada entitas produk. Berikut penjelasan untuk setiap endpoint yang diimplementasikan oleh metode dalam ProductRepository.

1. Metode findAll(categoryId, name):

   Metode ini mengambil semua produk dengan opsi untuk memfilter berdasarkan categoryId dan name. Metode ini mengatur query untuk menyertakan model Categories dengan alias category dan atribut name. Jika categoryId diberikan, produk yang dikembalikan akan difilter berdasarkan categoryId. Jika name diberikan, produk yang dikembalikan akan difilter berdasarkan name. Produk yang ditemukan akan diubah menjadi objek JavaScript sederhana dengan nama kategori ditambahkan sebagai properti categoryName, dan properti category akan dihapus. Produk kemudian dikembalikan dalam urutan berdasarkan createdAt secara menurun.

2. Metode findById(id):

   Metode ini mengambil satu produk berdasarkan ID yang diberikan. Produk yang ditemukan akan menyertakan model Categories dengan alias category dan atribut name. Metode ini mengembalikan produk lengkap dengan informasi kategori terkait.

3. Metode create(data):

   Metode ini digunakan untuk membuat produk baru. Data produk yang diterima sebagai parameter akan digunakan untuk membuat entri baru dalam tabel Products. Metode ini mengembalikan produk yang baru dibuat.

4. Metode update(product, data):

   Metode ini memperbarui produk yang ada dengan data baru yang diterima sebagai parameter. Produk yang akan diperbarui diterima sebagai parameter

pertama, dan data baru diterima sebagai parameter kedua. Metode ini mengembalikan produk yang telah diperbarui.

Metode delete(product):

Metode ini menghapus produk yang diterima sebagai parameter. Produk yang akan dihapus diterima sebagai parameter. Metode ini mengembalikan hasil penghapusan produk dari basis data.

g. Service

```
const productRepository =
require('../repositories/productRepository');
const { uploadImageToCloudinary } =
require('../middleware/upload');

class ProductService {
  async getAllProducts(categoryId, name) {
    return await productRepository.findAll(categoryId, name);
  }

  async getProductById(id) {
    const product = await productRepository.findById(id);
    if (!product) {
      throw new Error('Product not found');
    }
    return product;
  }

  async createProduct(data, image, user) {
    const imageUrl = await uploadImageToCloudinary(image);

    const productData = {
      ...data,
      imageUrl,
      createdBy: user.username,
      updatedBy: user.username,
    };

    const product = await productRepository.create(productData);
```

```
    return {
      id: product.id,
      name: product.name,
      qty: product.qty,
      categoryId: product.categoryId,
      imageUrl: product.imageUrl,
      createdAt: product.createdAt,
      updatedAt: product.updatedAt,
      createdBy: product.createdBy,
      updatedBy: product.updatedBy,
    };
  }

  async updateProduct(id, data, image, user) {
    const product = await productRepository.findById(id);
    if (!product) {
      throw new Error('Product not found');
    }

    if (image) {
      const imageUrl = await uploadImageToCloudinary(image);
      data.imageUrl = imageUrl;
    }

    data.updatedBy = user.username;

    return await productRepository.update(product, data);
  }

  async deleteProduct(id) {
    const product = await productRepository.findById(id);
    if (!product) {
      throw new Error('Product not found');
    }
    return await productRepository.delete(product);
  }
```

```
}

module.exports = new ProductService();
```

Kode di atas mendefinisikan kelas ProductService yang berfungsi sebagai lapisan layanan untuk mengelola produk dalam aplikasi. Kelas ini menggunakan productRepository untuk berinteraksi dengan basis data dan uploadImageToCloudinary untuk mengunggah gambar produk. Setiap metode dalam kelas ini menangani logika bisnis yang terkait dengan produk. Berikut penjelasan untuk setiap endpoint yang diimplementasikan oleh metode dalam ProductService.

1. Endpoint getAllProducts(categoryId, name):

   Metode ini mengambil semua produk dengan opsi untuk memfilter berdasarkan categoryId dan name. Metode ini memanggil productRepository.findAll(categoryId, name) untuk mengambil daftar produk dari basis data. Jika categoryId atau name diberikan, produk akan difilter sesuai dengan parameter yang diberikan.
2. Endpoint getProductById(id):

   Metode ini mengambil satu produk berdasarkan ID yang diberikan. Metode ini memanggil productRepository.findById(id) untuk menemukan produk di basis data. Jika produk tidak ditemukan, metode ini akan melemparkan error dengan pesan "Product not found". Jika produk ditemukan, produk tersebut akan dikembalikan.
3. Endpoint createProduct(data, image, user):

   Metode ini membuat produk baru. Data produk diterima sebagai parameter bersama dengan gambar dan informasi pengguna yang membuat produk. Gambar diunggah ke Cloudinary menggunakan uploadImageToCloudinary(image), dan URL gambar yang dihasilkan ditambahkan ke data produk. Informasi pengguna (createdBy dan updatedBy) juga ditambahkan ke data produk. Metode ini kemudian memanggil productRepository.create(productData) untuk membuat produk di basis data dan mengembalikan informasi produk yang baru dibuat.
4. Endpoint updateProduct(id, data, image, user):

   Metode ini memperbarui produk yang ada berdasarkan ID yang diberikan. Metode ini memanggil productRepository.findById(id) untuk menemukan produk di basis data. Jika produk tidak ditemukan, metode ini akan

melemparkan error dengan pesan "Product not found". Jika gambar baru diberikan, gambar tersebut akan diunggah ke Cloudinary dan URL gambar yang dihasilkan akan ditambahkan ke data produk. Informasi pengguna (updatedBy) juga diperbarui. Metode ini kemudian memanggil productRepository.update(product, data) untuk memperbarui produk di basis data dan mengembalikan produk yang telah diperbarui.

5. Endpoint deleteProduct(id):

   Metode ini menghapus produk berdasarkan ID yang diberikan. Metode ini memanggil productRepository.findById(id) untuk menemukan produk di basis data. Jika produk tidak ditemukan, metode ini akan melemparkan error dengan pesan "Product not found". Jika produk ditemukan, metode ini akan memanggil productRepository.delete(product) untuk menghapus produk dari basis data.

Front-End

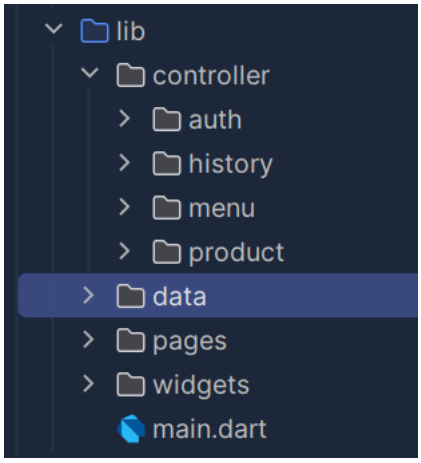
Tools

1. **Android Studio**

Library

1. **cupertino_icons**
2. **fluttertoast**
3. **http**
4. **image_picker**
5. **mime**
6. **http_parser**
7. **shared_preferences**
8. **lottie**
9. **get**
10. **shimmer**
11. **intl**

Arsitektur Folder



Penjelasan Code

Main.dart

```dart
void main() async {
  await initializeDateFormatting();
  runApp(MyApp());
}
```

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return GetMaterialApp(
      title: 'Your App',
      theme: ThemeData(
        primaryColor: Colors.blue,
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.blue),
        useMaterial3: true,
      ),
      debugShowCheckedModeBanner: false,
      initialRoute: '/splash',
      getPages: [
        GetPage(name: '/splash', page: () => SplashScreen(
          child: LoginScreen(),
          lottieUrl: 'https://lottie.host/87f13f4d-1435-4454-ae4d-
33d831f688b2/95vbhGRbrm.json',
        )),
        GetPage(name: '/login', page: () => LoginScreen()),
        GetPage(name: '/register', page: () => RegisterScreen()),
        GetPage(name: '/main', page: () => MainScreen()),
        GetPage(name: '/home', page: () => HomeScreen()),

        GetPage(name: '/product', page: () {
          var args = Get.arguments;
          return ProductsPage(
            categoryId: args['categoryId'],
            categoryName: args['categoryName'],
          );
        }),
        GetPage(name: '/detail-product', page: () {
          var args = Get.arguments;
          return ProductDetailPage(productId: args['productId']);
        }),
      ],
      localizationsDelegates: [
        DefaultMaterialLocalizations.delegate,
        DefaultWidgetsLocalizations.delegate,
      ],
      supportedLocales: [
        const Locale('en', ''),
```

```
        const Locale('id', ''),
      ],
    );
  }
}
```

Program di atas adalah aplikasi Flutter yang menggunakan beberapa paket termasuk get untuk navigasi dan manajemen status, serta intl untuk pengaturan format tanggal lokal. Pada fungsi main, format tanggal diinisialisasi dengan initializeDateFormatting sebelum aplikasi diluncurkan dengan menjalankan MyApp.

Kelas MyApp adalah widget Stateless yang mengembalikan GetMaterialApp sebagai root aplikasi. GetMaterialApp ini mengatur beberapa properti seperti title, theme, dan debugShowCheckedModeBanner. Aplikasi ini juga mendefinisikan rute awal (/splash) dan daftar rute lainnya dengan menggunakan getPages, yang mencakup halaman splash, login, register, main, home, products, dan detail products. Setiap rute diatur dengan menggunakan GetPage yang mendefinisikan nama rute dan widget yang akan ditampilkan saat rute tersebut diakses.

Di dalam getPages, beberapa halaman menerima argumen menggunakan Get.arguments, seperti pada halaman ProductsPage dan ProductDetailPage, yang memerlukan categoryId, categoryName, dan productId. Selain itu, aplikasi mendukung dua bahasa lokal yaitu Bahasa Inggris (en) dan Bahasa Indonesia (id), dengan mengatur localizationsDelegates dan supportedLocales untuk mendukung lokal materi default dan widget lokal.

Implementasi Consume API

1. Register Controller

```
class RegisterController extends GetxController {
  final usernameController = TextEditingController();
  final passwordController = TextEditingController();
  final confirmPasswordController = TextEditingController();
  var avatar = Rxn<File>();
  var isLoading = false.obs;
  String? avatarMimeType;

  final picker = ImagePicker();

  Future<void> pickImage() async {
```

```
    final pickedFile = await picker.pickImage(source:
ImageSource.gallery);

    if (pickedFile != null) {
      avatar.value = File(pickedFile.path);
      avatarMimeType = lookupMimeType(pickedFile.path);
      print('Selected file MIME type: $avatarMimeType');
    } else {
      print('No image selected.');
    }
  }

  Future<void> register() async {
    if (usernameController.text.isEmpty ||
        passwordController.text.isEmpty ||
        confirmPasswordController.text.isEmpty ||
        avatar.value == null) {
      _showAlertDialog('Error', 'Please fill in all fields and
select an image.');
      return;
    }

    if (passwordController.text != confirmPasswordController.text)
{
      _showAlertDialog('Error', 'Passwords do not match.');
      return;
    }

    isLoading.value = true;

    try {
      final url = Uri.parse('$apiUrl/users/register');
      final request = http.MultipartRequest('POST', url);
      request.fields['username'] = usernameController.text;
      request.fields['password'] = passwordController.text;

      if (avatar.value != null) {
        request.files.add(await http.MultipartFile.fromPath(
          'avatar',
          avatar.value!.path,
          contentType: MediaType.parse(avatarMimeType ??
```

```
'image/jpeg'),
        ));
      }

      request.headers['Content-Type'] = 'multipart/form-data';

      HttpClient httpClient = HttpClient()
        ..badCertificateCallback = (X509Certificate cert, String
host, int port) => true;
      IOClient ioClient = IOClient(httpClient);

      print('Sending request to $url');
      print('Request fields: ${request.fields}');
      print('Request files: ${request.files}');

      final streamedResponse = await ioClient.send(request);
      final response = await
http.Response.fromStream(streamedResponse);

      print('Response status: ${response.statusCode}');
      print('Response body: ${response.body}');

      final decodedData = jsonDecode(response.body);

      if (response.statusCode == 201) {
        _showAlertDialog('Success', decodedData['message'],
success: true);
      } else {
        _showAlertDialog('Error', decodedData['message']);
      }
    } catch (e) {
      _showAlertDialog('Error', 'An error occurred. Please try
again.');
      print(e.toString());
    } finally {
      isLoading.value = false;
    }
  }

  void _showAlertDialog(String title, String message, {bool success
= false}) {
```

```
    Get.defaultDialog(
      title: title,
      content: Text(message),
      textConfirm: 'OK',
      onConfirm: () {
        Get.back();
        if (success) {
          Get.offAllNamed('/login');
        }
      },
    );
  }
}
```

Program di atas adalah sebuah controller untuk fitur registrasi pengguna dalam aplikasi Flutter menggunakan paket GetX untuk manajemen state dan dependensi, serta paket http untuk mengirim permintaan HTTP. RegisterController memiliki beberapa properti dan metode untuk mengelola proses registrasi, termasuk pengambilan gambar avatar pengguna dan pengiriman data registrasi ke server.

Controller ini mengelola beberapa TextEditingController untuk mengumpulkan input pengguna seperti username, password, dan confirmPassword. Metode pickImage menggunakan ImagePicker untuk memungkinkan pengguna memilih gambar dari galeri, menyimpan file yang dipilih, dan menampilkan tipe MIME dari gambar tersebut. Metode register melakukan validasi input, memastikan semua bidang telah diisi dan kata sandi sesuai, sebelum mengirim permintaan HTTP POST menggunakan MultipartRequest untuk mengirimkan data pengguna dan gambar avatar ke endpoint registrasi.

Permintaan HTTP ini menggunakan IOClient yang dikonfigurasi untuk mengabaikan sertifikat yang tidak valid (dalam pengembangan). Setelah permintaan dikirim, respons diterima dan diproses. Jika pendaftaran berhasil, dialog konfirmasi ditampilkan, dan pengguna diarahkan ke halaman login. Jika ada kesalahan, pesan kesalahan akan ditampilkan dalam dialog. Controller ini juga menggunakan reaktifitas GetX (Rxn dan obs) untuk mengelola perubahan state dan menampilkan dialog menggunakan Get.defaultDialog.

2. Login Controller

```dart
class LoginController extends GetxController {
  var isLoading = false.obs;

  final usernameController = TextEditingController();
  final passwordController = TextEditingController();

  Future<void> login() async {
    if (usernameController.text.isEmpty ||
passwordController.text.isEmpty) {
      _showAlertDialog('Error', 'Please fill in all fields.');
      return;
    }

    try {
      isLoading(true);
      final url = Uri.parse('$apiUrl/users/login');

      HttpClient httpClient = HttpClient()
        ..badCertificateCallback = (X509Certificate cert, String
host, int port) => true;
      IOClient ioClient = IOClient(httpClient);

      final response = await ioClient.post(
        url,
        body: {
          'username': usernameController.text,
          'password': passwordController.text,
        },
      );

      print('Response status: ${response.statusCode}');
      print('Response body: ${response.body}');

      if (response.statusCode == 200) {
        final decodedData = jsonDecode(response.body);
        print('Login successful, saving token...');

        final prefs = await SharedPreferences.getInstance();
        final token = decodedData['data']['token'];
        await prefs.setString('token', token);
```

```
        final storedToken = prefs.getString('token');
        print('Stored token: $storedToken');

        showToast(message: "User is successfully signed in");

        Get.offAllNamed('/main');
      } else {
        final decodedData = jsonDecode(response.body);
        _showAlertDialog('Error', decodedData['message'] ?? 'Login
failed.');
      }
    } catch (e) {
      _showAlertDialog('Error', 'An error occurred. Please try
again.');
      print('Error during login: $e');
    } finally {
      isLoading(false);
    }
  }

  void _showAlertDialog(String title, String message) {
    Get.defaultDialog(
      title: title,
      middleText: message,
      textConfirm: 'OK',
      onConfirm: () {
        Get.back();
      },
    );
  }
}
```

Program di atas adalah sebuah controller untuk fitur login dalam aplikasi Flutter yang menggunakan GetX untuk manajemen state dan dependensi, serta SharedPreferences untuk menyimpan data secara lokal. LoginController memiliki beberapa properti dan metode untuk mengelola proses login, termasuk validasi input dan pengiriman data login ke server.

Controller ini mengelola TextEditingController untuk mengumpulkan input pengguna seperti username dan password. Metode login melakukan validasi input, memastikan

bahwa semua bidang telah diisi sebelum mengirim permintaan HTTP POST menggunakan IOClient yang telah dikonfigurasi untuk mengabaikan sertifikat yang tidak valid (untuk pengembangan).

Jika permintaan berhasil dan server mengembalikan status 200, token yang diterima dari server akan disimpan di SharedPreferences. Pengguna kemudian diarahkan ke halaman utama (/main). Jika login gagal, pesan kesalahan akan ditampilkan menggunakan dialog.

Selain itu, terdapat metode _showAlertDialog untuk menampilkan pesan kesalahan kepada pengguna. Controller ini menggunakan reaktifitas GetX (obs) untuk mengelola perubahan state seperti status pemuatan (isLoading) dan menampilkan toast notifikasi menggunakan showToast.

3. Category Controller

```
class HomeController extends GetxController {
  var categories = <dynamic>[].obs;
  var isLoading = true.obs;

  @override
  void onInit() {
    super.onInit();
    fetchCategories();
  }

  Future<void> fetchCategories() async {
    try {
      final prefs = await SharedPreferences.getInstance();
      final token = prefs.getString('token');

      print('Fetched token: $token');

      if (token == null) {
        navigateToLogin();
        return;
      }

      final url = Uri.parse('$apiUrl/categories');
```

```
      HttpClient httpClient = HttpClient()
        ..badCertificateCallback = (X509Certificate cert, String
host, int port) => true;
      IOClient ioClient = IOClient(httpClient);

      final response = await ioClient.get(
        url,
        headers: {
          HttpHeaders.authorizationHeader: 'Bearer $token',
        },
      );

      print('Response status: ${response.statusCode}');
      print('Response body: ${response.body}');

      if (response.statusCode == 200) {
        final decodedData = jsonDecode(response.body);
        categories.value = decodedData['data'];
        isLoading.value = false;
      } else {
        navigateToLogin();
      }
    } catch (e) {
      print('Error fetching categories: $e');
      navigateToLogin();
    }
  }

  void navigateToLogin() {
    Get.offAll(() => LoginScreen());
  }
}
```

Program di atas adalah sebuah controller untuk fitur beranda dalam aplikasi Flutter yang menggunakan GetX untuk manajemen state dan dependensi, serta SharedPreferences untuk menyimpan data secara lokal. HomeController mengelola data kategori dan status pemuatan (isLoading), serta bertanggung jawab untuk mengambil data kategori dari server ketika diinisialisasi.

Controller ini memiliki daftar categories yang bersifat reaktif (obs) untuk menyimpan data kategori yang diambil dari server. Metode fetchCategories digunakan untuk mengambil data kategori dengan mengirimkan permintaan HTTP GET ke endpoint yang telah ditentukan ($apiUrl/categories). Sebelum mengirim permintaan, metode ini mengambil token dari SharedPreferences dan menambahkannya ke header permintaan sebagai otorisasi.

Jika token tidak ditemukan, pengguna akan diarahkan ke halaman login. Jika permintaan berhasil dan server mengembalikan status 200, data kategori yang diterima akan disimpan dalam categories. Jika permintaan gagal atau terjadi kesalahan, pengguna juga akan diarahkan ke halaman login.

Selain itu, terdapat metode navigateToLogin untuk mengarahkan pengguna ke halaman login menggunakan GetX. Metode ini dipanggil ketika token tidak valid atau terjadi kesalahan selama pengambilan data kategori.

Secara keseluruhan, controller ini mengatur proses autentikasi pengguna dan pengambilan data kategori, serta memastikan bahwa pengguna yang tidak terautentikasi diarahkan ke halaman login.

4. History Controller

```
class HistoryController extends GetxController {
  var historyList = [].obs;
  var isLoading = true.obs;

  @override
  void onInit() {
    super.onInit();
    fetchHistory();
  }

  void fetchHistory() async {
    try {
      final prefs = await SharedPreferences.getInstance();
      final token = prefs.getString('token');

      if (token == null) {
        Get.offAllNamed('/login');
```

```
        return;
      }

      final response = await http.get(
        Uri.parse('$apiUrl/products'),
        headers: {
          'Authorization': 'Bearer $token',
        },
      );

      if (response.statusCode == 200) {
        final decodedData = jsonDecode(response.body);
        historyList.value = decodedData['data'];
      } else {
        print('Failed to fetch history');
      }
    } catch (e) {
      print('Failed to fetch history: $e');
    } finally {
      isLoading.value = false;
    }
  }
}
```

Program di atas adalah sebuah controller bernama HistoryController untuk mengelola data riwayat produk dalam aplikasi Flutter yang menggunakan GetX untuk manajemen state dan dependensi, serta SharedPreferences untuk menyimpan data secara lokal. HistoryController bertanggung jawab untuk mengambil dan menyimpan daftar riwayat produk yang pernah diakses oleh pengguna yang sedang masuk.

Controller ini memiliki dua variabel reaktif:

1. historyList: Menyimpan daftar riwayat produk yang diambil dari server.
2. isLoading: Menandakan status pemuatan data, apakah sedang dalam proses pengambilan data atau tidak.

Ketika controller diinisialisasi (onInit), metode fetchHistory akan dipanggil untuk mengambil data riwayat produk dari server. Proses pengambilan data meliputi:

1. Mengambil token otorisasi yang disimpan di SharedPreferences.

2. Jika token tidak ditemukan, mengarahkan pengguna ke halaman login menggunakan GetX.
3. Mengirim permintaan HTTP GET ke endpoint $apiUrl/products dengan token sebagai header otorisasi.
4. Jika permintaan berhasil (status kode 200), mengupdate variabel historyList dengan data yang diterima dari server.
5. Jika permintaan gagal, mencetak pesan kesalahan di konsol.
6. Mengubah status isLoading menjadi false setelah proses pengambilan data selesai, baik berhasil maupun gagal.

Secara keseluruhan, controller ini mengelola pengambilan dan penyimpanan data riwayat produk, memastikan bahwa hanya pengguna yang terautentikasi yang dapat mengakses data tersebut, dan menyediakan status pemuatan untuk memberikan umpan balik kepada antarmuka pengguna.

5. User Controller

```
class UserController extends GetxController {
  var username = ''.obs;
  var avatarPath = ''.obs;
  var isLoading = false.obs;

  @override
  void onInit() {
    super.onInit();
    fetchUserProfile();
  }

  void fetchUserProfile() async {
    isLoading(true);
    try {
      final prefs = await SharedPreferences.getInstance();
      final token = prefs.getString('token');

      if (token == null) {
        Get.offAll(() => LoginScreen());
        return;
      }
```

```
      final response = await http.get(
        Uri.parse('$apiUrl/users/current-user'),
        headers: {
          'Authorization': 'Bearer $token',
        },
      );

      if (response.statusCode == 200) {
        final data = json.decode(response.body)['data'];
        username(data['username']);
        avatarPath(data['avatarPath']);
      } else {
        Get.snackbar('Error', 'Failed to fetch user profile');
      }
    } catch (e) {
      Get.snackbar('Error', 'Failed to fetch user profile');
    } finally {
      isLoading(false);
    }
  }

  void logout() async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.remove('token');
    Get.offAll(() => LoginScreen());
  }
}
```

Program di atas adalah sebuah controller untuk mengelola profil pengguna dalam aplikasi Flutter yang menggunakan GetX untuk manajemen state dan dependensi, serta SharedPreferences untuk menyimpan data secara lokal. UserController bertanggung jawab untuk mengambil data profil pengguna yang sedang masuk (username dan avatarPath) dan mengelola status pemuatan (isLoading).

Controller ini memiliki dua variabel reaktif (username dan avatarPath) untuk menyimpan data pengguna dan variabel isLoading untuk mengindikasikan status pemuatan. Metode fetchUserProfile digunakan untuk mengambil data profil pengguna dari server ketika controller diinisialisasi.

Selama proses pengambilan profil, fetchUserProfile akan:

1. Mengambil token dari SharedPreferences.
2. Jika token tidak ditemukan, mengarahkan pengguna ke halaman login.
3. Mengirim permintaan HTTP GET ke endpoint $apiUrl/users/current-user dengan token sebagai header otorisasi.
4. Jika permintaan berhasil (status kode 200), mengupdate variabel username dan avatarPath dengan data yang diterima dari server.
5. Jika permintaan gagal, menampilkan snackbar error.

Selain itu, terdapat metode logout yang menghapus token dari SharedPreferences dan mengarahkan pengguna ke halaman login menggunakan GetX.

Secara keseluruhan, controller ini mengatur proses pengambilan data profil pengguna yang sedang masuk, memastikan pengguna yang tidak terautentikasi diarahkan ke halaman login, dan menyediakan fungsi logout untuk keluar dari aplikasi.

6. Product Controller

```
class ProductsController extends GetxController {
  var products = [].obs;
  var isLoading = true.obs;

  @override
  void onInit() {
    super.onInit();
  }

  Future<void> fetchProducts(int categoryId) async {
    try {
      final prefs = await SharedPreferences.getInstance();
      final token = prefs.getString('token');

      if (token == null) {
        Get.offAllNamed('/login');
        return;
      }

      final url =
```

```
Uri.parse('$apiUrl/products?categoryId=$categoryId');

      HttpClient httpClient = HttpClient()
        ..badCertificateCallback = (X509Certificate cert, String
host, int port) => true;
      IOClient ioClient = IOClient(httpClient);

      final response = await ioClient.get(
        url,
        headers: {
          HttpHeaders.authorizationHeader: 'Bearer $token',
        },
      );

      if (response.statusCode == 200) {
        final decodedData = jsonDecode(response.body);
        products.value = decodedData['data'];
        print(decodedData);
      } else {
        Get.offAllNamed('/login');
      }
    } catch (e) {
      Get.offAllNamed('/login');
      print('Error during fetching products: $e');
    } finally {
      isLoading.value = false;
    }
  }

  Future<void> searchProducts(int categoryId, String productName)
async {
    try {
      final prefs = await SharedPreferences.getInstance();
      final token = prefs.getString('token');

      if (token == null) {
        Get.offAllNamed('/login');
        return;
      }

      final url =
```

```
Uri.parse('$apiUrl/products?categoryId=$categoryId&name=$productNam
e');

      HttpClient httpClient = HttpClient()
        ..badCertificateCallback = (X509Certificate cert, String
host, int port) => true;
      IOClient ioClient = IOClient(httpClient);

      final response = await ioClient.get(
        url,
        headers: {
          HttpHeaders.authorizationHeader: 'Bearer $token',
        },
      );

      if (response.statusCode == 200) {
        final decodedData = jsonDecode(response.body);
        products.value = decodedData['data'];
        print(decodedData);
      } else {
        Get.offAllNamed('/login');
      }
    } catch (e) {
      Get.offAllNamed('/login');
      print('Error during fetching products: $e');
    } finally {
      isLoading.value = false;
    }
  }

}
```

Program di atas adalah sebuah controller bernama ProductsController untuk mengelola data produk dalam aplikasi Flutter yang menggunakan GetX untuk manajemen state dan dependensi, serta SharedPreferences untuk menyimpan data secara lokal. ProductsController bertanggung jawab untuk mengambil dan mencari daftar produk dari server berdasarkan kategori dan nama produk.

Controller ini memiliki dua variabel reaktif:

1. products: Menyimpan daftar produk yang diambil dari server.

2. isLoading: Menandakan status pemuatan data, apakah sedang dalam proses pengambilan data atau tidak.

Ketika controller diinisialisasi (onInit), tidak ada tindakan khusus yang dilakukan, tetapi metode fetchProducts dan searchProducts disediakan untuk mengambil dan mencari produk.

Metode fetchProducts melakukan hal-hal berikut:

1. Mengambil token otorisasi yang disimpan di SharedPreferences.
2. Jika token tidak ditemukan, mengarahkan pengguna ke halaman login menggunakan GetX.
3. Mengirim permintaan HTTP GET ke endpoint $apiUrl/products dengan parameter categoryId dan token sebagai header otorisasi.
4. Jika permintaan berhasil (status kode 200), mengupdate variabel products dengan data yang diterima dari server.
5. Jika permintaan gagal, mengarahkan pengguna ke halaman login.
6. Mengubah status isLoading menjadi false setelah proses pengambilan data selesai.

Metode searchProducts melakukan hal yang mirip dengan fetchProducts, tetapi juga menambahkan parameter name untuk mencari produk berdasarkan nama selain kategori.

Secara keseluruhan, controller ini mengelola pengambilan dan pencarian data produk, memastikan bahwa hanya pengguna yang terautentikasi yang dapat mengakses data tersebut, dan menyediakan status pemuatan untuk memberikan umpan balik kepada antarmuka pengguna.

7. Product Detail Controller

```
class ProductDetailController extends GetxController {
  var product = {}.obs;
  var isLoading = true.obs;

  Future<void> fetchProduct(int productId) async {
    try {
      final prefs = await SharedPreferences.getInstance();
      final token = prefs.getString('token');
```

```
      if (token == null) {
        _navigateToLogin();
        return;
      }

      final url = Uri.parse('$apiUrl/products/$productId');

      final response = await http.get(
        url,
        headers: {
          HttpHeaders.authorizationHeader: 'Bearer $token',
        },
      );

      if (response.statusCode == 200) {
        final decodedData = jsonDecode(response.body);
        product.value = decodedData['data'];
        isLoading.value = false;
        print(decodedData);
      } else {
        _navigateToLogin();
      }
    } catch (e) {
      _navigateToLogin();
      print('Error during fetching product: $e');
    }
  }

  Future<bool> deleteProduct(int productId) async {
    try {
      final prefs = await SharedPreferences.getInstance();
      final token = prefs.getString('token');

      if (token == null) {
        _navigateToLogin();
        return false;
      }

      final url = Uri.parse('$apiUrl/products/$productId');

      final response = await http.delete(
```

```
        url,
        headers: {
          HttpHeaders.authorizationHeader: 'Bearer $token',
        },
      );

      if (response.statusCode == 200) {
        return true;
      } else {
        // Handle error
        print('Error: ${response.statusCode}');
        return false;
      }
    } catch (e) {
      _navigateToLogin();
      print('Error during deleting product: $e');
      return false;
    }
  }

  Future<void> updateProduct(int productId, String name, int qty,
int categoryId, File? image) async {
    try {
      final prefs = await SharedPreferences.getInstance();
      final token = prefs.getString('token');

      if (token == null) {
        _navigateToLogin();
        return;
      }

      final uri = Uri.parse('$apiUrl/products/$productId');
      final request = http.MultipartRequest('PUT', uri)
        ..headers['Authorization'] = 'Bearer $token'
        ..fields['name'] = name
        ..fields['qty'] = qty.toString()
        ..fields['categoryId'] = categoryId.toString();

      if (image != null) {
        request.files.add(await http.MultipartFile.fromPath(
          'image',
```

```
          image.path,
          contentType: MediaType('image', 'jpeg'),
        ));
      }

      final response = await request.send();

      if (response.statusCode == 200) {
        Get.back();
      } else {
        // Handle error
        print('Error: ${response.statusCode}');
      }
    } catch (e) {
      _navigateToLogin();
      print('Error during updating product: $e');
    }
  }

  void _navigateToLogin() {
    Get.offAll(LoginScreen());
  }
}
```

Program di atas mendefinisikan ProductDetailController, sebuah controller menggunakan GetX dalam aplikasi Flutter, yang bertugas mengelola detail produk termasuk mengambil, menghapus, dan memperbarui produk. Controller ini memastikan hanya pengguna yang terautentikasi dapat melakukan operasi tersebut.

Metode dalam controller ini meliputi:

1. **fetchProduct**: Mengambil detail produk dari server berdasarkan productId yang diberikan. Jika token otorisasi tidak ditemukan, pengguna diarahkan ke halaman login. Jika permintaan berhasil, data produk disimpan dalam variabel product dan isLoading diatur ke false.
2. **deleteProduct**: Menghapus produk dari server berdasarkan productId yang diberikan. Jika token otorisasi tidak ditemukan, pengguna diarahkan ke halaman login. Jika penghapusan berhasil (status kode 200), metode mengembalikan nilai true; jika tidak, mengembalikan nilai false dan mencetak kesalahan.

3. **updateProduct**: Memperbarui detail produk di server berdasarkan productId, name, qty, categoryId, dan image yang diberikan. Jika token otorisasi tidak ditemukan, pengguna diarahkan ke halaman login. Jika gambar disertakan, gambar tersebut ditambahkan ke permintaan. Jika pembaruan berhasil (status kode 200), pengguna diarahkan kembali ke halaman sebelumnya.
4. **_navigateToLogin**: Mengarahkan pengguna ke halaman login jika token otorisasi tidak ditemukan atau terjadi kesalahan selama operasi.

Secara keseluruhan, ProductDetailController mengatur operasi CRUD (Create, Read, Update, Delete) untuk produk dengan memastikan otentikasi pengguna, menangani respons server, dan mengarahkan pengguna ke halaman login jika diperlukan.

8. Create Product Controller

```
class CreateProductController extends GetxController {
  Future<void> createProduct({
    required String name,
    required int qty,
    required int categoryId,
    File? image,
  }) async {
    try {
      final prefs = await SharedPreferences.getInstance();
      final token = prefs.getString('token');

      if (token == null) {
        Get.offAllNamed('/login');
        return;
      }

      final uri = Uri.parse('$apiUrl/products/');
      final request = http.MultipartRequest('POST', uri)
        ..fields['name'] = name
        ..fields['qty'] = qty.toString()
        ..fields['categoryId'] = categoryId.toString()
        ..headers['Authorization'] = 'Bearer $token';

      if (image != null) {
        request.files.add(await http.MultipartFile.fromPath(
```

```
          'image',
          image.path,
          contentType: MediaType('image', 'jpeg'),
        ));
      }

      final response = await request.send();

      if (response.statusCode == 201) {
        Get.back();
      } else {
        print('Error: ${response.statusCode}');
        // Tambahkan penanganan kesalahan sesuai kebutuhan
      }
    } catch (e) {
      Get.offAllNamed('/login');
      print('Error during creating product: $e');
    }
  }
}
```

Program di atas mendefinisikan sebuah controller bernama CreateProductController menggunakan framework GetX dalam aplikasi Flutter. Controller ini berfungsi untuk menangani logika pembuatan produk baru dengan mengirimkan data produk ke server melalui permintaan HTTP POST.

Metode utama dalam controller ini adalah createProduct, yang menerima parameter berikut:

1. name: Nama produk (required).
2. qty: Kuantitas produk (required).
3. categoryId: ID kategori produk (required).
4. image: Gambar produk dalam bentuk File (opsional).

Langkah-langkah yang dilakukan metode createProduct:

1. Mengambil token otorisasi yang disimpan di SharedPreferences. Token ini digunakan untuk mengotentikasi permintaan ke server.

2. Jika token tidak ditemukan, pengguna akan diarahkan ke halaman login menggunakan GetX.
3. Membuat URI untuk endpoint API yang digunakan untuk membuat produk baru.
4. Menginisialisasi http.MultipartRequest untuk mengirim permintaan POST dengan beberapa bidang yang diisi (name, qty, categoryId) dan menambahkan header otorisasi dengan token.
5. Jika gambar produk disertakan, gambar tersebut ditambahkan ke permintaan sebagai file multipart dengan tipe konten 'image/jpeg'.
6. Mengirim permintaan ke server.
7. Jika permintaan berhasil (status kode 201), pengguna akan diarahkan kembali ke halaman sebelumnya. Jika tidak, mencetak kesalahan dan dapat menambahkan penanganan kesalahan lebih lanjut sesuai kebutuhan.
8. Jika terjadi kesalahan selama proses permintaan, pengguna akan diarahkan ke halaman login dan kesalahan akan dicetak di konsol.

Secara keseluruhan, controller ini memastikan bahwa hanya pengguna yang terautentikasi dapat membuat produk baru, mengirim data produk dan gambar ke server, dan menangani respons dari server sesuai dengan status kode yang diterima.