Prompting

Buatkan aplikasi penjualan secangkir kopi disisi backend menggunakan ExpressJS dengan database MySQL, dimana database tersebut menampung CRUD semua table pembeli, kasir, dan transaksi penjualan kopi. Setiap table berelasi satu dengan yang lainnya sehingga dapat menghasilkan laporan setiap bulan. Folder dan file dipisahkan kedalam Models, Controller, dan Router yang memiliki komentar disetiap kode programnya.

---

**Panduan Lengkap Membangun Backend Aplikasi Penjualan Kopi dengan ExpressJS dan MySQL**

**I. Pendahuluan: Arsitektur dan Peta Jalan Aplikasi Penjualan Kopi**

**A. Visi Proyek: Membangun API yang Profesional dan Terstruktur**

Tujuan dari proyek ini melampaui sekadar menciptakan serangkaian *endpoint* yang fungsional. Visi utamanya adalah untuk membangun sebuah fondasi *backend* yang profesional, terstruktur, dan siap untuk berkembang. Aplikasi yang baik tidak hanya bekerja sesuai permintaan, tetapi juga mudah dipahami, dipelihara, dan diperluas di masa depan. Untuk mencapai tujuan ini, proyek akan mengadopsi pola arsitektur **Model-Controller-Router**, sebuah adaptasi dari konsep Model-View-Controller (MVC) yang sangat populer dan efektif untuk membangun RESTful API.

Pola ini didasarkan pada prinsip fundamental dalam rekayasa perangkat lunak, yaitu *Separation of Concerns* (Pemisahan Kepentingan). Dengan memisahkan aplikasi menjadi tiga komponen logis yang berbeda, setiap bagian memiliki tanggung jawab yang jelas dan independen :

1. **Model**: Bertanggung jawab secara eksklusif untuk semua interaksi dengan *database*. Lapisan ini mengelola data, menjalankan *query*, dan menangani logika bisnis yang terkait langsung dengan struktur data.

2. **Controller**: Bertindak sebagai "otak" aplikasi. Lapisan ini menerima permintaan yang diteruskan oleh Router, memproses data masukan, memanggil fungsi yang sesuai di Model untuk berinteraksi dengan *database*, dan kemudian memformat respons yang akan dikirim kembali ke klien.

3. **Router**: Berfungsi sebagai "gerbang masuk" atau mekanisme perutean aplikasi. Tanggung jawab utamanya adalah mendefinisikan URL *endpoint* dan metode HTTP (GET, POST, PUT, DELETE) yang terkait, lalu meneruskan setiap permintaan yang masuk ke fungsi Controller yang tepat untuk ditangani.

Dengan menerapkan arsitektur ini, proyek ini tidak hanya akan menghasilkan kode yang berfungsi, tetapi juga sebuah sistem yang modular, terorganisir dengan baik, dan lebih mudah untuk diuji dan dikelola dalam jangka panjang.

**B. Peta Jalan API: Rangkuman Endpoint yang Akan Dibangun**

Sebelum menulis satu baris kode pun, praktik rekayasa perangkat lunak yang baik menuntut adanya perencanaan dan desain yang matang. Salah satu langkah pertama adalah mendefinisikan "kontrak" API, yaitu daftar lengkap *endpoint* yang akan disediakan oleh *backend*. Peta jalan ini berfungsi sebagai spesifikasi fungsional, memberikan gambaran yang jelas tentang cakupan dan kapabilitas aplikasi.

Tabel di bawah ini merangkum semua *endpoint* yang akan dibangun dalam panduan ini. Ini bukan sekadar daftar isi, melainkan sebuah cetak biru yang akan memandu proses pengembangan dan menjadi referensi krusial bagi siapa pun yang akan mengintegrasikan API ini di masa depan, misalnya pengembang *frontend*. Dengan mendefinisikan ekspektasi secara eksplisit di awal, proses pengembangan menjadi lebih terarah dan sistematis.

**Tabel 1: Peta Jalan Endpoint API Penjualan Kopi**

MappingEndpointAPI

**II. Fondasi Data: Perancangan dan Implementasi Skema Database MySQL**

Fondasi dari setiap aplikasi yang digerakkan oleh data adalah skema *database*-nya. Desain *database* yang baik bukan hanya tentang menyimpan data, tetapi juga tentang menegakkan aturan bisnis, memastikan integritas data, dan mengoptimalkan kinerja. Di bagian ini, kita akan merancang dan mengimplementasikan struktur *database* MySQL yang akan menjadi tulang punggung aplikasi penjualan kopi.

**A. Prinsip Desain Database Relasional**

Aplikasi ini akan menggunakan MySQL, sebuah Sistem Manajemen *Database* Relasional (RDBMS). Konsep inti dari RDBMS adalah data diorganisir ke dalam tabel-tabel yang saling berhubungan.

- **Tabel**: Entitas logis yang menyimpan data, mirip dengan *spreadsheet*. Kita akan memiliki tiga tabel utama: pembeli, kasir, dan transaksi.
- **Kolom (Atribut)**: Mendefinisikan jenis data yang disimpan dalam sebuah tabel, seperti nama_pembeli atau total_harga.
- **Primary Key (PK)**: Sebuah kolom (atau kombinasi kolom) yang secara unik mengidentifikasi setiap baris dalam sebuah tabel. Contohnya adalah id_pembeli di tabel pembeli.
- **Foreign Key (FK)**: Sebuah kolom dalam satu tabel yang merujuk ke *Primary Key* di tabel lain. FK adalah mekanisme yang menciptakan dan menegakkan hubungan antar tabel, memastikan bahwa data tetap konsisten. Misalnya, tabel

transaksi akan memiliki pembeli_id sebagai FK yang merujuk ke id_pembeli di tabel pembeli.

Visualisasi hubungan ini sering digambarkan dalam sebuah *Entity-Relationship Diagram* (ERD), yang berfungsi sebagai cetak biru arsitektur data kita. Dalam kasus ini, ERD akan menunjukkan bahwa satu

pembeli dapat memiliki banyak transaksi, dan satu kasir dapat menangani banyak transaksi. Ini adalah hubungan "satu-ke-banyak" (*one-to-many*).

**B. Skrip SQL untuk Pembuatan Tabel**

Skema *database* adalah sumber kebenaran absolut dan lapisan aturan paling kaku dalam aplikasi. Sementara kode aplikasi bisa mengandung *bug*, batasan seperti NOT NULL atau FOREIGN KEY di tingkat *database* bersifat mutlak. Dengan mendefinisikan aturan-aturan ini di *database*, kita membangun sistem yang lebih tangguh di mana korupsi data dapat dicegah secara aktif. Sebagai contoh, jika sebuah *bug* dalam kode mencoba membuat transaksi dengan pembeli_id yang tidak ada, *database* akan menolak operasi INSERT tersebut berkat adanya batasan *foreign key*, sehingga mencegah terciptanya data yatim (*orphaned record*).

Berikut adalah skrip SQL CREATE TABLE yang lengkap untuk membuat ketiga tabel yang dibutuhkan. Setiap kolom, tipe data, dan batasan telah dipilih dengan cermat untuk memastikan integritas dan efisiensi data.

**1. Tabel pembeli**

Tabel ini menyimpan informasi dasar mengenai setiap pelanggan atau pembeli.

SQL

```sql
-- Membuat tabel untuk menyimpan data pembeli
CREATE TABLE pembeli (
    -- ID unik untuk setiap pembeli, bertambah otomatis
    id_pembeli INT AUTO_INCREMENT PRIMARY KEY,
    -- Nama lengkap pembeli, tidak boleh kosong
    nama_pembeli VARCHAR(100) NOT NULL,
    -- Nomor telepon pembeli, bisa digunakan untuk kontak
    nomor_telepon VARCHAR(15),
    -- Alamat email pembeli, harus unik untuk setiap pembeli
    email VARCHAR(100) UNIQUE,
    -- Timestamp kapan data pembeli ini pertama kali dibuat
    dibuat_pada TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB;
```

**2. Tabel kasir**

Tabel ini menyimpan informasi mengenai setiap kasir yang beroperasi.

SQL

```sql
-- Membuat tabel untuk menyimpan data kasir
CREATE TABLE kasir (
    -- ID unik untuk setiap kasir, bertambah otomatis
    id_kasir INT AUTO_INCREMENT PRIMARY KEY,
    -- Nama lengkap kasir, tidak boleh kosong
    nama_kasir VARCHAR(100) NOT NULL,
    -- Nomor identifikasi pegawai untuk kasir, harus unik
    nomor_pegawai VARCHAR(50) UNIQUE NOT NULL,
    -- Timestamp kapan data kasir ini pertama kali dibuat
    dibuat_pada TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB;
```

**3. Tabel transaksi**

Ini adalah tabel inti yang mencatat setiap penjualan. Tabel ini menghubungkan pembeli dan kasir melalui *Foreign Keys*.

SQL

```sql
-- Membuat tabel untuk mencatat setiap transaksi penjualan
CREATE TABLE transaksi (
    -- ID unik untuk setiap transaksi, bertambah otomatis
    id_transaksi INT AUTO_INCREMENT PRIMARY KEY,
    -- Foreign Key yang merujuk ke ID pembeli di tabel 'pembeli'
    pembeli_id INT NOT NULL,
    -- Foreign Key yang merujuk ke ID kasir di tabel 'kasir'
    kasir_id INT NOT NULL,
    -- Tanggal dan waktu kapan transaksi terjadi
    tanggal_transaksi TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    -- Total harga dari transaksi, menggunakan tipe data DECIMAL untuk presisi keuangan
    total_harga DECIMAL(10, 2) NOT NULL,
    -- Metode pembayaran yang digunakan (misal: 'Tunai', 'Kartu Kredit', 'Digital')
    metode_pembayaran VARCHAR(50),

    -- Mendefinisikan batasan Foreign Key untuk pembeli_id
    -- Ini memastikan bahwa setiap transaksi harus terhubung dengan pembeli yang valid
    CONSTRAINT fk_pembeli
        FOREIGN KEY (pembeli_id)
        REFERENCES pembeli(id_pembeli)
        ON DELETE RESTRICT -- Mencegah penghapusan pembeli jika masih memiliki transaksi
        ON UPDATE CASCADE, -- Jika id_pembeli berubah, update juga di sini

    -- Mendefinisikan batasan Foreign Key untuk kasir_id
    -- Ini memastikan bahwa setiap transaksi harus ditangani oleh kasir yang valid
    CONSTRAINT fk_kasir
        FOREIGN KEY (kasir_id)
        REFERENCES kasir(id_kasir)
        ON DELETE RESTRICT -- Mencegah penghapusan kasir jika masih memiliki transaksi
        ON UPDATE CASCADE -- Jika id_kasir berubah, update juga di sini
) ENGINE=InnoDB;
```

**Penjelasan Batasan FOREIGN KEY:**

- ON DELETE RESTRICT: Aturan ini sangat penting untuk integritas data historis. Ini mencegah penghapusan data seorang pembeli atau kasir jika mereka masih tercatat dalam tabel transaksi. Untuk menghapus pembeli, semua transaksinya harus dihapus terlebih dahulu (atau diubah menjadi milik pembeli lain).
- ON UPDATE CASCADE: Jika (karena alasan tertentu) id_pembeli atau id_kasir di tabel induknya berubah, perubahan tersebut akan secara otomatis diperbarui di semua baris terkait di tabel transaksi. Ini menjaga konsistensi hubungan.

**III. Inisialisasi Proyek dan Konfigurasi Lingkungan ExpressJS**

Setelah fondasi data dirancang, langkah selanjutnya adalah mempersiapkan lingkungan pengembangan untuk *backend*. Bagian ini akan memandu proses pembuatan proyek Node.js, instalasi dependensi yang diperlukan, dan konfigurasi koneksi ke *database* MySQL.

**A. Memulai Proyek Node.js**

Proses ini dimulai dengan membuat direktori proyek dan menginisialisasi manajer paket Node.js (NPM).

1. **Buat Direktori Proyek**: Buka terminal atau *command prompt* Anda, navigasikan ke lokasi di mana Anda ingin menyimpan proyek, dan buat direktori baru.

   ```
   Bash

   mkdir aplikasi-kopi-backend
   cd aplikasi-kopi-backend
   ```

2. **Inisialisasi Proyek NPM**: Jalankan perintah berikut untuk membuat berkas package.json. Berkas ini akan melacak semua informasi proyek dan dependensinya.

   ```
   Bash

   npm init -y
   ```

3. **Instalasi Dependensi**: Kita memerlukan beberapa paket NPM untuk membangun aplikasi ini.

   1. express: Kerangka kerja web utama untuk Node.js.
   2. mysql2: Driver MySQL modern untuk Node.js yang mendukung *Promises*, membuatnya lebih mudah digunakan dengan sintaks async/await.
   3. nodemon: Alat pengembangan yang secara otomatis me-*restart* server setiap kali ada perubahan pada kode, mempercepat siklus pengembangan.
   4. dotenv: Modul yang memuat variabel lingkungan dari berkas .env, memungkinkan kita untuk memisahkan konfigurasi (seperti kredensial *database*) dari kode.

   Instal semua dependensi tersebut dengan perintah berikut:

   ```
   Bash

   npm install express mysql2 dotenv
   npm install nodemon --save-dev
   ```

4. **Konfigurasi Skrip NPM**: Buka berkas package.json dan tambahkan skrip start dan dev untuk memudahkan menjalankan server.

```
JSON
```

```
"scripts": {
  "start": "node server.js",
  "dev": "nodemon server.js"
},
```

Sekarang, server dapat dijalankan dalam mode produksi dengan npm start atau dalam mode pengembangan dengan npm run dev.

**B. Konfigurasi Server Utama dan Koneksi Database**

Dengan struktur proyek yang sudah siap, saatnya menulis kode *boilerplate* awal.

1. **Buat Berkas .env**: Di direktori utama proyek, buat berkas baru bernama .env. Berkas ini akan menyimpan kredensial *database* kita. Memisahkan informasi sensitif seperti ini adalah praktik keamanan yang krusial untuk mencegah kredensial terekspos di *repository* kode.

```
Cuplikan kode
```

```
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=password_anda
DB_NAME=nama_database_anda
PORT=3000
```

**Penting**: Ganti password_anda dan nama_database_anda dengan konfigurasi MySQL lokal Anda. Jangan lupa untuk menambahkan .env ke dalam berkas .gitignore Anda agar tidak terunggah ke Git.

2. **Buat Konfigurasi Database**: Untuk menjaga modularitas, logika koneksi *database* akan ditempatkan di berkas terpisah. Buat folder config dan di dalamnya buat berkas database.js.

JavaScript

```
// config/database.js

// Mengimpor paket dotenv untuk memuat variabel lingkungan dari file.env
require('dotenv').config();
// Mengimpor paket mysql2/promise untuk interaksi database dengan async/await
const mysql = require('mysql2/promise');

// Membuat connection pool untuk mengelola koneksi ke database MySQL secara efisien
// Pool lebih efisien daripada membuat koneksi baru untuk setiap query
const pool = mysql.createPool({
    host: process.env.DB_HOST,       // Host database dari variabel lingkungan
    user: process.env.DB_USER,       // User database dari variabel lingkungan
    password: process.env.DB_PASSWORD, // Password database dari variabel lingkungan
    database: process.env.DB_NAME,   // Nama database dari variabel lingkungan
    waitForConnections: true,        // Menunggu koneksi tersedia jika semua sedang digunakan
    connectionLimit: 10,             // Jumlah maksimum koneksi dalam pool
    queueLimit: 0                    // Tidak ada batasan antrian untuk query yang masuk
});

// Mengekspor pool agar bisa digunakan di modul lain (misalnya, di models)
module.exports = pool;
```

Pemisahan ini adalah penerapan dari *Single Responsibility Principle*. Berkas database.js hanya bertanggung jawab untuk konfigurasi dan penyediaan koneksi *database*, sementara berkas server utama akan fokus pada logika aplikasi web.

3. **Buat Server Express Utama**: Di direktori utama, buat berkas server.js. Ini adalah titik masuk utama aplikasi kita.

JavaScript

```
// server.js

// Mengimpor paket dotenv untuk memuat variabel lingkungan
require('dotenv').config();
// Mengimpor framework Express
const express = require('express');

// Membuat instance aplikasi Express
const app = express();
// Mengambil port dari variabel lingkungan, atau default ke 3000
const PORT = process.env.PORT |
```

| 3000;

```
// Middleware untuk mem-parsing body request dalam format JSON
// Ini memungkinkan kita untuk membaca data yang dikirim dari klien (misal: dari form)
app.use(express.json());

// Rute dasar untuk memastikan server berjalan
app.get('/', (req, res) => {
    res.send('Selamat Datang di API Penjualan Kopi!');
});

// Menjalankan server dan mendengarkan koneksi yang masuk pada port yang ditentukan
app.listen(PORT, () => {
    console.log(`Server berjalan pada port ${PORT}`);
});
```
Jalankan server dengan perintah `npm run dev`. Jika semua konfigurasi benar, Anda akan melihat pesan "Server berjalan

**IV. Implementasi Arsitektur Inti: Model, Controller, dan Router**

Dengan server dasar yang telah berjalan, kini saatnya untuk membangun kerangka arsitektur Model-Controller-Router. Struktur ini akan menjadi fondasi yang menopang seluruh logika aplikasi, memastikan setiap bagian kode memiliki tempat dan tujuan yang jelas.

**A. Membangun Struktur Folder Proyek**

Organisasi folder yang baik adalah kunci untuk menjaga proyek tetap terkelola seiring dengan pertumbuhannya. Berdasarkan pola arsitektur yang dipilih, kita akan membuat tiga direktori utama di dalam folder proyek :

- /models: Direktori ini akan berisi semua berkas yang bertanggung jawab untuk berinteraksi langsung dengan *database*. Setiap tabel dalam *database* (pembeli, kasir, transaksi) akan memiliki berkas modelnya sendiri.
- /controllers: Direktori ini akan menampung logika bisnis aplikasi. Setiap *resource* (pembeli, kasir, transaksi) akan memiliki berkas *controller*-nya sendiri yang berisi fungsi-fungsi untuk menangani permintaan seperti membuat, membaca, memperbarui, dan menghapus data.
- /routes: Direktori ini akan mendefinisikan *endpoint* API. Setiap *resource* akan memiliki berkas rutenya sendiri yang memetakan URL dan metode HTTP ke fungsi *controller* yang sesuai.

Struktur folder proyek Anda sekarang akan terlihat seperti ini:

```
aplikasi-kopi-backend/
├── config/
│   └── database.js
├── controllers/
├── models/
├── routes/
├── node_modules/
├──.env
├── package.json
├── package-lock.json
└── server.js
```

**B. Peran dan Tanggung Jawab Setiap Komponen**

Memahami alur permintaan melalui arsitektur ini sangat penting untuk proses pengembangan dan *debugging*. Alur ini bersifat searah dan logis: sebuah permintaan dari klien pertama kali diterima oleh Router, yang kemudian meneruskannya ke Controller yang sesuai. Controller memproses permintaan tersebut, memanggil fungsi dari Model untuk berinteraksi dengan *database* jika diperlukan. Model mengembalikan data ke Controller, yang kemudian memformatnya menjadi respons dan mengirimkannya kembali ke klien.

1. **Models (Lapisan Data)** Tanggung jawab utama lapisan Model adalah sebagai jembatan antara aplikasi dan *database*. Semua

*query* SQL mentah akan dienkapsulasi di dalam fungsi-fungsi pada lapisan ini. Lapisan Controller dan Router tidak perlu tahu detail implementasi *database*; mereka hanya perlu memanggil metode yang disediakan oleh Model, seperti Pembeli.findAll() atau Transaksi.create(data). Abstraksi ini membuat aplikasi lebih fleksibel; jika suatu saat kita memutuskan untuk beralih dari MySQL ke *database* lain, perubahan hanya perlu dilakukan di dalam lapisan Model tanpa mempengaruhi lapisan lainnya.

2. **Controllers (Lapisan Logika)** Controller adalah pusat pemrosesan logika. Setiap fungsi di dalam Controller menerima dua objek utama dari Express:

req (*request*) dan res (*response*). Fungsi ini akan:

- Mengekstrak data dari permintaan masuk (misalnya, dari req.body untuk data POST, atau req.params untuk ID dari URL).
- Melakukan validasi dasar pada data masukan.
- Memanggil satu atau lebih fungsi dari lapisan Model untuk mengambil atau memanipulasi data.
- Menangani *error* yang mungkin terjadi selama proses.
- Menyusun dan mengirimkan respons kembali ke klien menggunakan metode dari objek res (misalnya, res.status(200).json(data)).

3. **Routers (Lapisan Perutean)** Tanggung jawab lapisan Router sangat spesifik: mendefinisikan *endpoint* API. Berkas-berkas di dalam direktori

/routes akan menggunakan objek express.Router() untuk membuat modul rute yang dapat diekspor. Setiap rute mengikat sebuah metode HTTP (.get(), .post(), .put(), .delete()) dan sebuah pola URL (misalnya, /pembeli/:id) ke sebuah fungsi spesifik di dalam Controller. Lapisan ini tidak mengandung logika bisnis apa pun; tugasnya murni sebagai pemetaan dan delegasi.

**C. Menghubungkan Komponen di server.js**

Setelah membuat struktur folder, kita perlu mengintegrasikannya ke dalam aplikasi Express utama. Kita akan membuat satu berkas rute utama yang akan menggabungkan semua rute dari masing-masing *resource*.

1. **Buat Berkas Rute Utama**: Di dalam folder /routes, buat berkas index.js. Berkas ini akan berfungsi sebagai agregator untuk semua rute lainnya.

JavaScript

```
// routes/index.js

const express = require('express');
const router = express.Router();

// Di sini kita akan mengimpor rute-rute spesifik nanti
// const pembeliRoutes = require('./pembeliRoutes');
// router.use('/pembeli', pembeliRoutes);

module.exports = router;
```

2. **Hubungkan Rute Utama ke server.js**: Sekarang, modifikasi server.js untuk menggunakan berkas rute utama ini.

JavaScript

```
// server.js (versi yang diperbarui)

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

| 3000;

```
// Impor rute utama dari folder routes
const mainRouter = require('./routes/index');

app.use(express.json());

app.get('/', (req, res) => {
    res.send('Selamat Datang di API Penjualan Kopi!');
});

// Gunakan rute utama untuk semua permintaan yang masuk ke /api
// Ini berarti semua rute yang didefinisikan di dalam mainRouter akan diawali dengan /api
app.use('/api', mainRouter);

app.listen(PORT, () => {
    console.log(`Server berjalan pada port ${PORT}`);
});
```

Dengan konfigurasi ini, setiap permintaan yang masuk ke aplikasi kita dengan awalan /api akan ditangani oleh sistem perutean yang telah kita bangun. Misalnya, permintaan ke http://localhost:3000/api/pembeli akan diarahkan ke modul rute pembeli. Kerangka kerja arsitektural ini sekarang siap untuk diisi dengan fungsionalitas CRUD yang sebenarnya.

**V. Manajemen Data Master: Endpoint CRUD untuk Entitas Pembeli dan Kasir**

Pada bagian ini, kita akan mengimplementasikan fungsionalitas *Create, Read, Update, Delete* (CRUD) secara penuh untuk entitas data master, yaitu pembeli dan kasir. Kita akan menggunakan pembeli sebagai studi kasus mendalam untuk menjelaskan setiap langkah dalam arsitektur Model-Controller-Router. Pola yang sama kemudian akan diterapkan untuk entitas kasir.

**A. CRUD Lengkap untuk Pembeli (Studi Kasus Detail)**

Implementasi ini akan melibatkan pembuatan tiga berkas: pembeliModel.js, pembeliController.js, dan pembeliRoutes.js.

**1. Model (models/pembeliModel.js)**

Berkas ini berisi logika untuk berinteraksi dengan tabel pembeli di *database*. Setiap fungsi akan mengeksekusi satu *query* SQL. Penggunaan *parameterized queries* (tanda ?) adalah wajib untuk mencegah serangan *SQL Injection*.

pembeliModel.js

**2. Controller (controllers/pembeliController.js)**

Berkas ini menangani logika HTTP. Setiap fungsi di sini akan dipanggil oleh Router, kemudian memanggil fungsi yang sesuai di Model, dan akhirnya mengirimkan respons JSON ke klien dengan status HTTP yang benar.

pembeliController.js

**3. Router (routes/pembeliRoutes.js)**

Berkas ini mendefinisikan *endpoint* untuk *resource* pembeli dan menghubungkannya ke fungsi-fungsi di pembeliController.

pembeliRoutes.js

**4. Mengintegrasikan Rute Pembeli**

Terakhir, kita perlu mendaftarkan pembeliRoutes di berkas rute utama (routes/index.js).

JavaScript

```
// routes/index.js (versi yang diperbarui)

const express = require('express');
const router = express.Router();

// Mengimpor rute pembeli
const pembeliRoutes = require('./pembeliRoutes');

// Menggunakan rute pembeli untuk semua permintaan yang diawali dengan /pembeli
// Contoh: GET /api/pembeli akan ditangani oleh pembeliRoutes
router.use('/pembeli', pembeliRoutes);

module.exports = router;
```

**B. CRUD Lengkap untuk Kasir (Penerapan Pola)**

Struktur kode untuk CRUD kasir akan sangat mirip dengan pembeli. Kemiripan ini bukanlah sebuah kebetulan, melainkan bukti keberhasilan arsitektur yang telah kita pilih. Pola yang konsisten mempercepat pengembangan dan membuat *codebase* lebih mudah diprediksi dan dipahami.

Berikut adalah kode lengkap untuk Model, Controller, dan Router kasir.

1. Model (models/kasirModel.js)

kasirModel.js

2. Controller (controllers/kasirController.js)

kasirController.js

3. Router (routes/kasirRoutes.js)

kasirRoutes.js

4. Mengintegrasikan Rute Kasir

Terakhir, daftarkan kasirRoutes di routes/index.js.

Index.js

Dengan ini, *endpoint* CRUD untuk data master pembeli dan kasir telah selesai diimplementasikan dengan mengikuti arsitektur yang bersih dan terstruktur.

**VI. Inti Operasional: Endpoint CRUD untuk Entitas Transaksi Penjualan**

Setelah membangun fondasi untuk data master, kini kita beralih ke inti operasional aplikasi: manajemen transaksi. Entitas transaksi lebih kompleks karena ia menghubungkan data dari tabel pembeli dan kasir. Implementasi CRUD untuk transaksi akan menunjukkan bagaimana arsitektur kita menangani relasi antar data dan menyajikan informasi yang lebih kaya dan kontekstual.

**A. Membuat dan Membaca Data Transaksi**

Fokus utama di sini adalah pada operasi create dan read. Operasi read akan menjadi lebih menarik karena kita akan menggunakan JOIN dalam *query* SQL untuk mengambil data terkait dari tabel lain, sebuah praktik yang sangat efisien.

**1. Model (models/transaksiModel.js)**

Model transaksi akan berisi *query* yang lebih kompleks. Khususnya, fungsi findAll dan findById akan menggunakan LEFT JOIN untuk menggabungkan data dari tabel pembeli dan kasir.

Penggunaan JOIN di lapisan model adalah keputusan desain yang krusial. Alternatifnya adalah mengambil data transaksi terlebih dahulu, lalu melakukan *query* terpisah untuk setiap pembeli dan kasir. Pendekatan ini, yang dikenal sebagai masalah "*N+1 query*", sangat tidak efisien dan akan membebani *database* seiring bertambahnya jumlah data. Dengan JOIN, kita dapat mengambil semua informasi yang dibutuhkan dalam satu kali perjalanan ke *database*, yang secara dramatis meningkatkan performa aplikasi.

transaksiModel.js

**2. Controller (controllers/transaksiController.js)**

Controller untuk transaksi akan mengikuti pola yang sama, menangani req dan res, serta memanggil metode dari transaksiModel.

transaksiController.js

**B. Memperbarui dan Menghapus Transaksi**

Melengkapi siklus CRUD, kita sekarang akan menambahkan fungsi untuk memperbarui dan menghapus data transaksi.

**1. Melengkapi Model (models/transaksiModel.js)**

Tambahkan fungsi update dan deleteById ke objek Transaksi di transaksiModel.js.

transaksiModel.js

**2. Melengkapi Controller (controllers/transaksiController.js)**

Tambahkan *handler* updateTransaksi dan deleteTransaksi ke transaksiController.js.

transaksiController.js

**3. Router (routes/transaksiRoutes.js) dan Integrasi**

Buat berkas routes/transaksiRoutes.js dan daftarkan di routes/index.js.

transaksiRoutes.js

Index.js

Dengan ini, seluruh fungsionalitas CRUD untuk entitas transaksi telah selesai, menunjukkan bagaimana arsitektur yang solid dapat dengan mudah mengakomodasi relasi dan penyajian data yang kompleks.

**VII. Puncak Fungsionalitas: Rute Laporan Penjualan Bulanan**

Ini adalah bagian di mana semua komponen yang telah kita bangun—tabel yang saling berelasi dan arsitektur yang terstruktur—bersatu untuk menghasilkan nilai bisnis yang nyata: laporan. Fitur ini akan menunjukkan kekuatan SQL dalam melakukan agregasi data dan bagaimana ExpressJS dapat menyajikannya sebagai *endpoint* API yang berguna untuk analisis atau *dashboard*.

Fitur ini juga menyoroti prinsip penting: delegasikan pekerjaan komputasi data yang kompleks ke *database*. *Database* relasional seperti MySQL dirancang dan dioptimalkan secara khusus untuk operasi seperti JOIN, GROUP BY, dan fungsi agregat (SUM, COUNT). Mencoba melakukan agregasi serupa di dalam kode JavaScript (dengan mengambil semua data transaksi lalu melakukan *looping*) akan jauh lebih lambat, memakan lebih banyak memori, dan lebih rentan terhadap kesalahan.

**A. Merancang Query SQL untuk Agregasi Data**

Inti dari fitur laporan ini adalah satu *query* SQL yang kuat. Mari kita pecah *query* ini menjadi beberapa bagian untuk memahami logikanya.

SQL

```sql
SELECT
    -- 1. Ekstrak tahun dari tanggal transaksi
    YEAR(t.tanggal_transaksi) AS tahun,
    -- 2. Ekstrak nama bulan dari tanggal transaksi
    MONTHNAME(t.tanggal_transaksi) AS bulan,
    -- 3. Hitung jumlah total transaksi dalam grup (bulan) ini
    COUNT(t.id_transaksi) AS jumlah_transaksi,
    -- 4. Jumlahkan total harga dari semua transaksi dalam grup ini
    SUM(t.total_harga) AS total_pendapatan
FROM
    transaksi AS t -- 5. Mulai dari tabel transaksi
GROUP BY
    -- 6. Kelompokkan hasil berdasarkan tahun dan bulan
    YEAR(t.tanggal_transaksi),
    MONTH(t.tanggal_transaksi),
    MONTHNAME(t.tanggal_transaksi)
ORDER BY
    -- 7. Urutkan hasil berdasarkan tahun dan bulan secara kronologis
    tahun ASC,
    MONTH(t.tanggal_transaksi) ASC;
```

**Penjelasan Langkah demi Langkah:**

1. YEAR(t.tanggal_transaksi): Fungsi SQL yang mengekstrak komponen tahun dari kolom tanggal_transaksi.
2. MONTHNAME(t.tanggal_transaksi): Fungsi yang mengekstrak nama bulan (misalnya, 'January', 'February'). Ini lebih mudah dibaca daripada nomor bulan.
3. COUNT(t.id_transaksi): Fungsi agregat yang menghitung jumlah baris (transaksi) dalam setiap grup.
4. SUM(t.total_harga): Fungsi agregat yang menjumlahkan nilai dari kolom total_harga untuk semua transaksi dalam setiap grup.
5. FROM transaksi AS t: Menentukan bahwa sumber data utama kita adalah tabel transaksi, dengan alias t untuk mempersingkat.
6. GROUP BY...: Ini adalah bagian terpenting. Perintah ini menginstruksikan *database* untuk mengelompokkan semua baris yang memiliki nilai tahun dan bulan yang sama ke dalam satu baris ringkasan. Fungsi agregat (COUNT dan SUM) kemudian beroperasi pada setiap grup ini.
7. ORDER BY...: Mengurutkan hasil akhir agar laporan disajikan secara kronologis, yang lebih intuitif untuk dibaca.

**B. Implementasi Endpoint Laporan**

Sekarang kita akan mengimplementasikan *query* ini ke dalam arsitektur Model-Controller-Router kita.

**1. Model (models/laporanModel.js)**

Kita akan membuat berkas model baru yang khusus menangani *query* terkait laporan.

laporanModel.js

**2. Controller (controllers/laporanController.js)**

Controller ini akan sangat sederhana, karena semua logika kompleks sudah ditangani oleh *database*. Tugasnya hanya memanggil model dan mengirimkan hasilnya.

laporanController.js

**3. Router (routes/laporanRoutes.js)**

Terakhir, kita buat berkas rute untuk *endpoint* laporan.

laporanRoutes.js

**4. Integrasi Final**

Daftarkan laporanRoutes di routes/index.js.

Index.js

Dengan ini, *endpoint* GET /api/laporan/bulanan sekarang aktif dan siap menyajikan data agregat penjualan yang berharga, menyelesaikan fitur inti dari aplikasi *backend* ini.

**VIII. Standar Profesional: Dokumentasi Kode Komprehensif dengan JSDoc**

Menulis kode yang berfungsi adalah satu hal, tetapi menulis kode yang dapat dipahami, dipelihara, dan digunakan oleh orang lain (atau diri Anda sendiri di masa depan) adalah tanda seorang profesional. Salah satu pilar utama dari kode berkualitas adalah dokumentasi. Alih-alih menggunakan komentar biasa (// atau /* */), kita akan menerapkan standar **JSDoc**, sebuah bahasa *markup* yang memungkinkan kita untuk membuat anotasi pada kode JavaScript secara terstruktur.

**A. Pengenalan JSDoc: Komentar sebagai Dokumentasi**

JSDoc bukan sekadar komentar; ini adalah komentar yang dapat dibaca oleh mesin. Dengan mengikuti sintaks JSDoc, kita dapat:

- **Menghasilkan Dokumentasi HTML**: Alat JSDoc dapat memindai kode sumber dan secara otomatis menghasilkan situs web dokumentasi API yang lengkap.

- **Meningkatkan Bantuan IDE**: *Editor* kode modern seperti Visual Studio Code dapat membaca anotasi JSDoc untuk memberikan fitur *IntelliSense* (pelengkapan otomatis), informasi tipe, dan deskripsi parameter yang kaya, bahkan dalam proyek JavaScript biasa.
- **Mendorong Desain yang Lebih Baik**: Proses menulis dokumentasi JSDoc memaksa pengembang untuk berpikir secara eksplisit tentang "kontrak" sebuah fungsi: apa saja parameternya, tipe datanya, apa yang dihasilkannya, dan apa tujuannya. Proses ini sering kali mengungkap kelemahan dalam desain atau logika sebelum kode tersebut dijalankan.

**B. Praktik Terbaik: Menerapkan JSDoc di Seluruh Proyek**

Mari kita lihat bagaimana menerapkan JSDoc pada beberapa fungsi yang telah kita buat. Komentar JSDoc selalu dimulai dengan /** dan diakhiri dengan */.

**1. Contoh JSDoc untuk Fungsi Model**

Model berinteraksi langsung dengan *database* dan sering kali mengembalikan *Promise*. Dokumentasi harus mencerminkan hal ini.

JavaScript

```
// models/pembeliModel.js (dengan JSDoc)

const pool = require('../config/database');

const Pembeli = {
    /**
     * Mencari satu pembeli berdasarkan ID-nya.
     * Fungsi ini menjalankan query SELECT untuk mendapatkan satu record dari tabel pembeli.
     * @param {number} id - ID unik dari pembeli yang akan dicari.
     * @returns {Promise<object|null>} Sebuah Promise yang resolve dengan objek pembeli jika ditemu
     */
    findById: async (id) => {
        const query = 'SELECT * FROM pembeli WHERE id_pembeli =?';
        const [rows] = await pool.query(query, [id]);
        return rows |

| null;
    },
};
```

- @param {number} id: Mendokumentasikan parameter id, menyatakan tipenya adalah number dan memberikan deskripsi.
- @returns {Promise<object|null>}: Mendokumentasikan nilai kembali. Ini menyatakan bahwa fungsi mengembalikan sebuah Promise yang, ketika berhasil, akan berisi sebuah object (data pembeli) atau null.

**2. Contoh JSDoc untuk Fungsi Controller**

Fungsi *controller* di Express memiliki tanda tangan standar (req, res). Kita dapat menggunakan fitur import() dari JSDoc untuk mereferensikan tipe data dari pustaka Express, memberikan bantuan pengetikan yang sangat akurat.

```
// controllers/pembeliController.js (dengan JSDoc)

const Pembeli = require('../models/pembeliModel');

const pembeliController = {
    /**
     * Handler untuk mendapatkan pembeli berdasarkan ID.
     * Mengambil ID dari parameter URL, memanggil model untuk mencari data,
     * dan mengirimkan respons JSON yang sesuai.
     * @param {import('express').Request} req - Objek request Express. Berisi parameter URL (req.para
     * @param {import('express').Response} res - Objek response Express. Digunakan untuk mengirim res
     * @returns {void} Fungsi ini tidak mengembalikan nilai secara langsung, tetapi mengirimkan respo
     */
    getPembeliById: async (req, res) => {
        try {
            const id = req.params.id;
            const pembeli = await Pembeli.findById(id);
            if (pembeli) {
                res.status(200).json(pembeli);
            } else {
                res.status(404).json({ message: 'Pembeli tidak ditemukan' });
            }
        } catch (error) {
            res.status(500).json({ message: 'Error mendapatkan data pembeli', error: error.message })
        }
    },
};
```

- @param {import('express').Request} req: Anotasi ini memberi tahu IDE bahwa parameter req adalah objek Request dari Express. Ini akan mengaktifkan pelengkapan otomatis untuk properti seperti req.params, req.body, dll.
- @returns {void}: Menunjukkan bahwa fungsi ini tidak memiliki nilai return yang bermakna; tugasnya adalah menyelesaikan siklus permintaan-respons dengan mengirimkan respons.

Dengan menerapkan JSDoc secara konsisten di seluruh basis kode, kita mengubah komentar dari sekadar catatan menjadi alat pengembangan yang aktif, meningkatkan kejelasan, kualitas, dan kemudahan pemeliharaan proyek secara signifikan.

**IX. Pengujian, Kesimpulan, dan Arah Pengembangan Selanjutnya**

Setelah menyelesaikan implementasi kode, langkah terakhir adalah memastikan semuanya berfungsi seperti yang diharapkan dan merencanakan potensi pengembangan di masa depan. Bagian ini memberikan panduan singkat untuk pengujian API dan menyajikan beberapa ide untuk meningkatkan aplikasi lebih lanjut.

**A. Menguji API Menggunakan Postman**

Karena *backend* kita tidak memiliki antarmuka pengguna grafis, kita memerlukan alat khusus untuk berinteraksi dengan *endpoint* API yang telah kita buat. Postman adalah salah satu alat paling populer untuk tujuan ini.

Berikut adalah panduan singkat untuk menguji beberapa *endpoint* utama:

1. **Membuat Pembeli Baru (POST /api/pembeli)**
    - Buka Postman dan buat permintaan baru.
    - Ubah metode HTTP menjadi POST.
    - Masukkan URL: http://localhost:3000/api/pembeli.
    - Pilih *tab* Body, lalu pilih opsi raw dan set tipenya menjadi JSON.
    - Masukkan data pembeli baru dalam format JSON di area teks :

JSON

```
{
    "nama_pembeli": "Budi Santoso",
    "nomor_telepon": "081234567890",
    "email": "budi.s@example.com"
}
```

1.
    - Klik Send. Anda seharusnya menerima respons dengan status 201 Created dan pesan sukses.
2. **Mendapatkan Semua Pembeli (GET /api/pembeli)**
    - Buat permintaan baru atau ubah yang sudah ada.
    - Set metode HTTP menjadi GET.
    - Masukkan URL: http://localhost:3000/api/pembeli.
    - Klik Send. Anda akan melihat daftar semua pembeli (termasuk Budi Santoso yang baru saja Anda tambahkan) dalam format JSON dengan status 200 OK.
3. **Mendapatkan Laporan Bulanan (GET /api/laporan/bulanan)**
    - Setelah Anda membuat beberapa transaksi (menggunakan *endpoint* POST /api/transaksi), Anda dapat menguji *endpoint* laporan.
    - Set metode HTTP menjadi GET.

- Masukkan URL: http://localhost:3000/api/laporan/bulanan.
- Klik Send. Anda akan menerima respons JSON yang mengelompokkan total penjualan per bulan.

Ulangi proses ini untuk menguji semua *endpoint* lain (PUT, DELETE, dan GET by ID) untuk memastikan seluruh fungsionalitas CRUD berjalan dengan benar.

**B. Rangkuman dan Pembelajaran Utama**

Melalui panduan ini, kita telah membangun lebih dari sekadar aplikasi; kita telah menerapkan prinsip-prinsip rekayasa perangkat lunak yang penting untuk pengembangan *backend* yang solid:

- **Pemisahan Kepentingan (Separation of Concerns)**: Arsitektur Model-Controller-Router memastikan bahwa setiap bagian kode memiliki tanggung jawab tunggal, membuat aplikasi lebih mudah dipahami, di-debug, dan dikembangkan.
- **Desain Berbasis Data**: Kita memulai dengan merancang skema *database* yang kuat, menggunakan *primary* dan *foreign keys* untuk menegakkan integritas data pada tingkat yang paling fundamental.
- **Alur Data Searah**: Permintaan dari klien mengikuti alur yang dapat diprediksi melalui Router, Controller, dan Model, yang menyederhanakan logika dan penelusuran kesalahan.
- **Delegasi Tugas yang Tepat**: Tugas-tugas yang membutuhkan banyak komputasi data, seperti agregasi untuk laporan, didelegasikan ke *database* MySQL, yang jauh lebih efisien untuk pekerjaan tersebut daripada JavaScript di sisi server.

**C. Langkah Berikutnya: Peningkatan Aplikasi**

Aplikasi yang telah kita bangun adalah fondasi yang kokoh. Berikut adalah beberapa area di mana aplikasi ini dapat ditingkatkan lebih lanjut, yang dapat menjadi tantangan belajar berikutnya:

1. **Validasi Input**: Saat ini, aplikasi kita mempercayai semua data yang dikirim oleh klien. Di dunia nyata, ini sangat berbahaya. Implementasikan pustaka validasi seperti Joi atau express-validator di lapisan Controller untuk memastikan bahwa data yang masuk (misalnya, req.body) memiliki format dan tipe yang benar sebelum diproses lebih lanjut.
2. **Penanganan Error Terpusat**: Alih-alih menggunakan blok try...catch di setiap fungsi *controller*, buatlah sebuah *middleware* penanganan error terpusat di Express. Ini akan menangkap semua *error* yang terjadi di aplikasi Anda dalam satu tempat, menyederhanakan kode dan memungkinkan *logging* error yang konsisten.
3. **Keamanan (Authentication & Authorization)**: Saat ini, semua *endpoint* bersifat publik. Langkah selanjutnya yang krusial adalah menambahkan sistem otentikasi untuk mengidentifikasi pengguna (misalnya, kasir yang *login*) dan otorisasi untuk mengontrol akses mereka. Teknologi

umum untuk ini adalah *JSON Web Tokens* (JWT), yang memungkinkan Anda untuk mengamankan *endpoint* sehingga hanya pengguna yang terotentikasi dan berwenang yang dapat mengaksesnya.

4. **Paginasi**: Untuk *endpoint* yang mengembalikan daftar data (seperti GET /api/pembeli atau GET /api/transaksi), jika jumlah data menjadi sangat besar, mengirim semuanya sekaligus akan menjadi tidak efisien. Implementasikan paginasi menggunakan parameter *query* LIMIT dan OFFSET di SQL untuk mengembalikan data dalam potongan-potongan yang lebih kecil.

Dengan melanjutkan pengembangan pada area-area ini, Anda akan terus memperdalam pemahaman Anda tentang pengembangan *backend* dan membangun aplikasi yang lebih kuat, aman, dan siap untuk produksi.