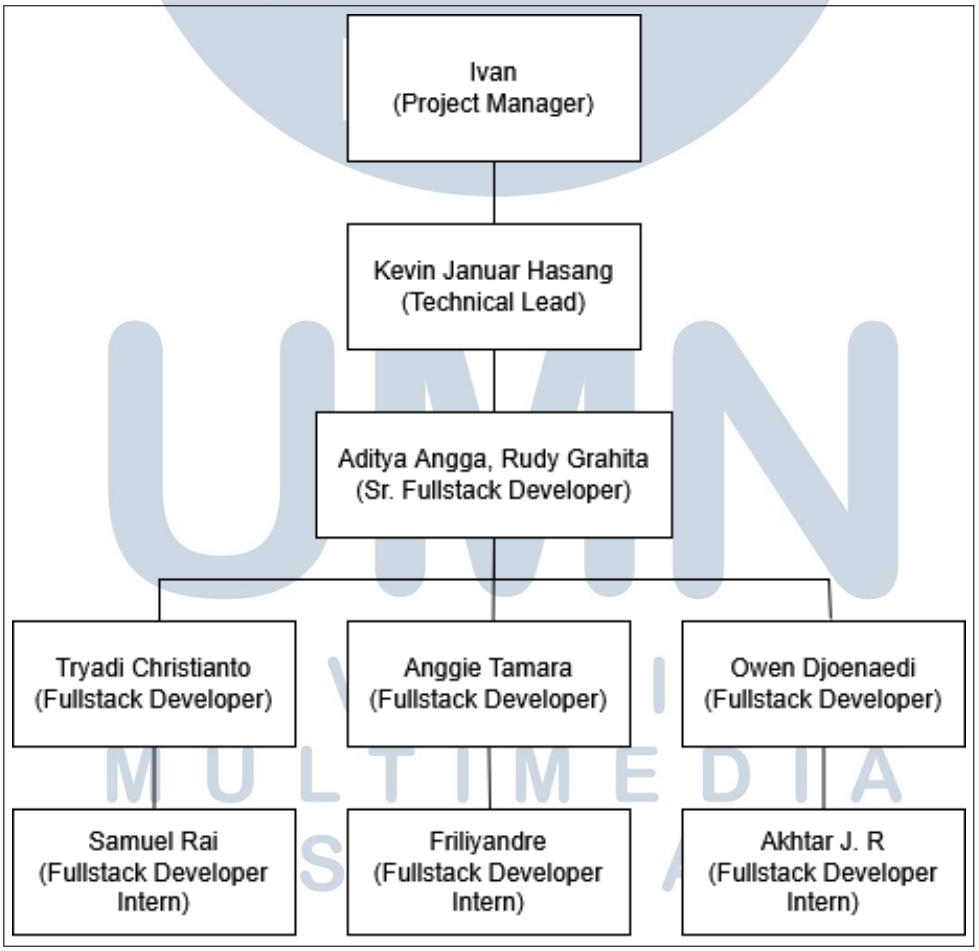
BAB 3
PELAKSANAAN KERJA MAGANG

3.1 Kedudukan dan Koordinasi

Kedudukan selama melakukan proses magang dalam PT Magna Solusi Indonesia adalah sebagai *Fullstack Developer*. Posisi ini dijalankan di bawah bimbingan para senior *developer* dan *technical lead* seperti pada Gambar 3.1. Tugas utama yang diberikan yaitu berperan sebagai *support* dalam pengembangan aplikasi *Distribution Management System* (DMS) untuk salah satu klien kami. Secara spesifik, tanggung jawab tersebut mencakup pembuatan *master dashboard*, integrasi antara *front-end* dan *back-end*, serta perbaikan *(fixing) unit test* guna memastikan performa aplikasi.



Gambar 3.1. Struktur Proyek DMS

Koordinasi dilakukan dengan cara melakukan evaluasi secara rutin terhadap setiap komponen dan fitur aplikasi, baik secara tatap muka maupun daring. Evaluasi tersebut tidak hanya dilakukan dengan sesama *developer*, namun juga oleh beberapa tim *analyst* serta *project manager*, guna memastikan kesesuaian dari segi bisnis, tampilan dan fungsi. Apabila terdapat kendala dalam proses *development*, komunikasi dengan *developer* yang lebih senior dilakukan untuk mendapatkan solusi secara tepat dan menghemat waktu, sehingga pekerjaan dapat berjalan secara *on-track* dan tepat sasaran.

3.2 Tugas yang Dilakukan

Selama pelaksanaan kerja magang, tugas yang dilakukan dapat dilihat pada Tabel 3.1:

Minggu Ke -	Pekerjaan yang dilakukan
1	Memahami *boilerplate front-end* dan *back-end* proyek
2	Membuat *front-end* dan CRUD *dashboard master unit of measurement*
3	Membuat tampilan dan CRUD *dashboard master unit of measurement conversion* serta konfigurasi LoV (*List of Values*)
4	Membuat *front-end monitoring claim*
5	Memperbaiki *master location LoV, mandatory field, & input field*
6	Mengintegrasikan Jaspersoft Studio dengan aplikasi untuk mencetak *report* dalam bentuk PDF dalam dashboard *Delivery Order*
7	Menerapkan *Filtering & pagination* data
8	Menerapkan *checkbox* pada master *territory*
9	Memperbaiki master item
10	Membuat tampilan *Sales Order* & memperbaiki LoV

Tabel 3.1. Pekerjaan yang dilakukan tiap minggu selama pelaksanaan kerja magang

3.3 Uraian Pelaksanaan Magang

Berikut adalah uraian dari pelaksanaan pekerjaan magang yang telah disebutkan sebelumnya.

3.3.1 Memahami Boilerplate Front-End dan Back-End Proyek

Dalam proses *development* yang menggunakan *Java Spring Boot* dan *AngularJS* sebagai *framework*. diperlukan beberapa pemahaman mengenai struktur dari kedua *framework* itu sendiri. Pada *Java Spring Boot*, hal-hal yang perlu diketahui adalah:

1. Desain dan membuat struktur model data (Model, VO/DTO) yang merepresentasikan entitas yang digunakan pada tabel. VO biasanya digunakan untuk menyimpan data yang tidak akan berubah selama masa hidup objek tersebut, sedangkan DTO digunakan untuk mengirim data antar layer aplikasi dan bisa diubah saat proses transfer.

2. Menyusun DAO *(Data Access Object)* dan DAO *(Implementation)* untuk berinteraksi langsung dengan basis data, menggunakan *query* yang disesuaikan dengan kebutuhan fungsional, seperti contohnya *filtering & pagination*.

3. Mengimplementasikan *service layer* sebagai logika yang mengatur alur validasi data, seperti yang paling umum digunakan adalah saat ingin menambah data, memodifikasi, dan menghapus data.

4. Membuat RESTful API pada *controller* untuk menghubungkan antara logika *back-end* dan *front-end*.

5. Pada aplikasi DMS, sebagian besar proses dalam mendapatkan data, menambahkan data, dan memodifikasi data akan menggunakan metode POST. Hal tersebut digunakan untuk menangani pengiriman data yang kompleks seperti *filter, limit, offset, sort*, dan *search*. POST memungkinkan *body request* mengandung objek JSON. Jika menggunakan GET, maka data-data yang kompleks harus dikodekan sebagai URL parameter yang panjang.

Kemudian pada *AngularJS*, hal-hal yang perlu diketahui adalah:

1. Menggunakan API yang telah dibuat pada *back-end* untuk melakukan *trigger* kepada *database* saat menjalankan perintah-perintah yang langsung berhubungan dengan data.

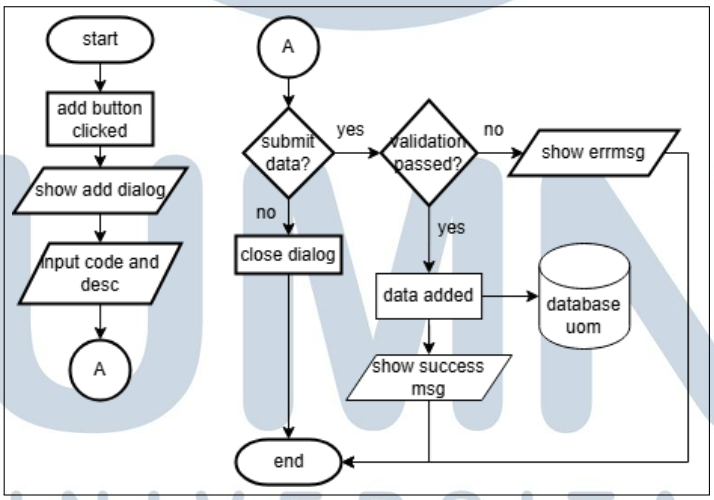
2. Mengembangkan dan menggunakan ulang antarmuka pengguna berbasis komponen seperti *form, filter, search, multi-select dropdown*, dan lain sebagainya.

3. Menangani logika tampilan berbasis mode seperti *isAdd, isEdit, isDelete, isView, isSearch* agar halaman bersifat dinamis dan dapat digunakan sesuai konteks maupun kebutuhan.

3.3.2 Master Unit of Measurement dan Master Unit of Measurement Conversion

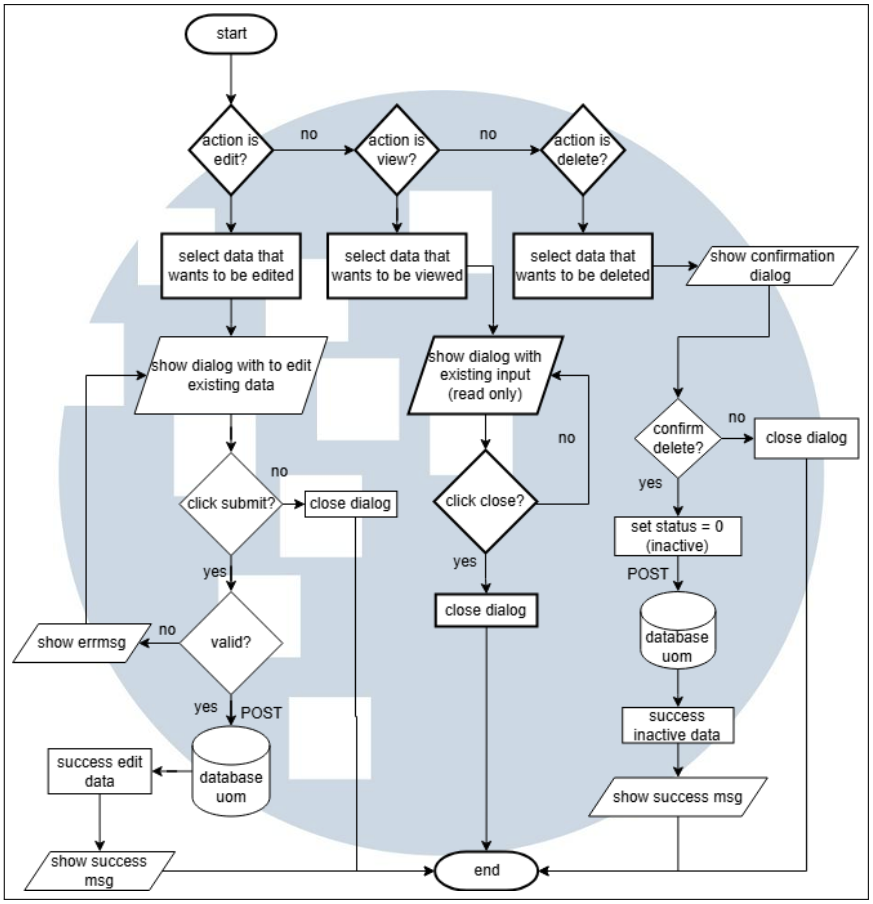
Tugas pertama yang dilakukan adalah membuat tampilan Master UoM serta konfigurasi dengan *back-end* agar data-data yang dikirim dikembalikan menjadi sebuah data yang dapat digunakan untuk keperluan master-master lain. Tugas kedua yang dilakukan yaitu membuat UoM *Conversion* juga terlihat serupa, hanya saja ada beberapa perubahan isi data dan penambahan fitur LoV *(List of Values)*.

Dashboard UoM berisi daftar satuan dasar seperti meter, liter, dan sebagainya. Sementara itu, UoM *Conversion* digunakan untuk menyimpan relasi konversi antar satuan, dengan mengacu pada `nid_uom_from` dan `nid_uom_to` sebagai *foreign key* untuk id dari UoM. Sebelum menuju penjelasan CRUD UoM, berikut adalah *flowchart* untuk memperjelas alur CRUD UoM.



Gambar 3.2. Flowchart Add UoM

Pada Gambar 3.2, dijelaskan alur untuk menambah data UoM. Pertama, *user* perlu untuk menekan tombol **Add New**, setelah itu akan muncul dialog untuk mengisi data kode dan deskripsi. Setelah diisi, akan dicek ke *back-end* apakah validasi lolos, jika iya maka data berhasil ditambahkan pada *database* dan tampil pesan sukses. Jika tidak, maka tampil pesan *error*.



Gambar 3.3. Flowchart Edit, View, dan Delete UoM

Pada Gambar 3.3 merupakan alur dari Edit, View, dan Delete pada Master UoM. Saat edit, akan dibuka dialog *field-field* yang telah terisi sebelumnya, lalu bisa langsung mengubah *field* yang diinginkan. Jika setelah dicek validasi sukses, maka akan tampil pesan berhasil edit data, jika tidak maka akan menampilkan pesan *error*. Saat view, hanya menampilkan data yang terpilih dengan *field disabled*, dan delete akan langsung menampilkan pesan konfirmasi, jika **"Yes"**, maka akan mengubah status dari 1 menjadi 0 dan jika **"No"** akan menutup pesan konfirmasi tersebut.

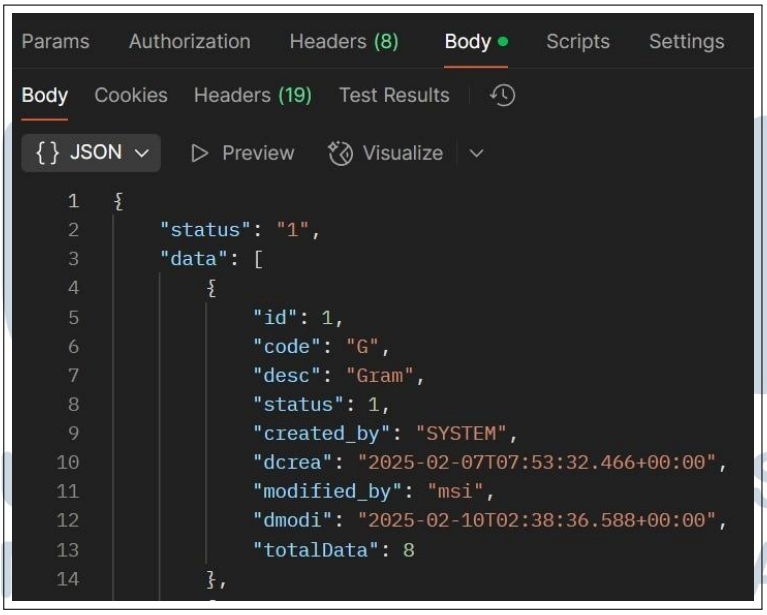
A Back-end pada Master Unit of Measurement

Dalam master ini, terdapat fitur-fitur seperti *search, reset, add, update / edit, view*, serta *delete (soft delete)*. Sebelum menuju ke tampilan, terdapat beberapa API yang dibuat untuk menjalankan fitur-fitur CRUD. *Endpoint* dari API tersebut dapat dilihat pada Tabel 3.2.

Tabel 3.2. Tabel endpoint pada UoM

Nama API	Endpoint	Metode	Deskripsi
Get All UoM	/	POST	API ini berfungsi untuk mendapatkan semua data UoM.
Add UoM	/add	POST	API ini berfungsi untuk membuat UoM baru.
Update UoM	/update/:updateid	POST	API ini berfungsi untuk memperbarui UoM yang diinginkan.
Soft Delete UoM	/update/:updateid	POST	API ini berfungsi untuk mengubah status pada UoM yang diinginkan.

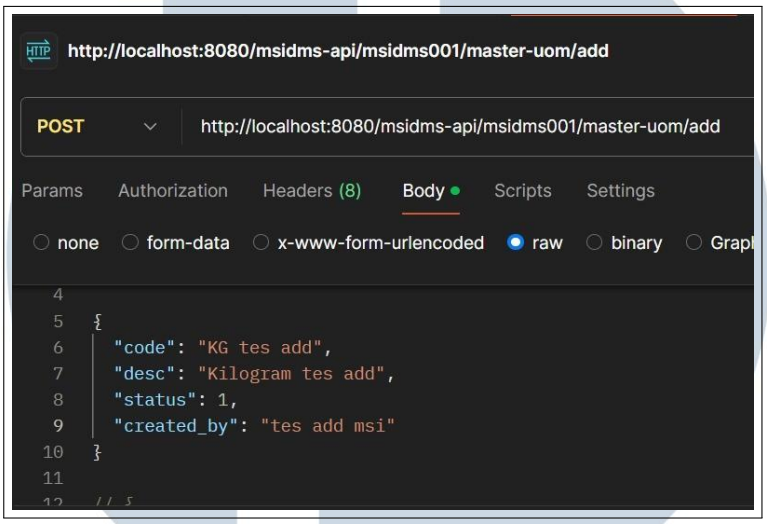
Setelah proses pembuatan API selesai, tahap selanjutnya adalah melakukan pengujian menggunakan Postman untuk memastikan API berjalan dengan baik dan siap untuk digunakan pada sisi *front-end*. Guna mengambil seluruh data UoM digunakan *endpoint* (*/*) dengan metode POST, yang berada di bawah *base URL* **/master-uom**, sehingga membentuk URL lengkap **/master-uom/**. Hasil pengujian ini dapat dilihat pada Gambar 3.4.



Gambar 3.4. *Get UoM Data* pada Postman

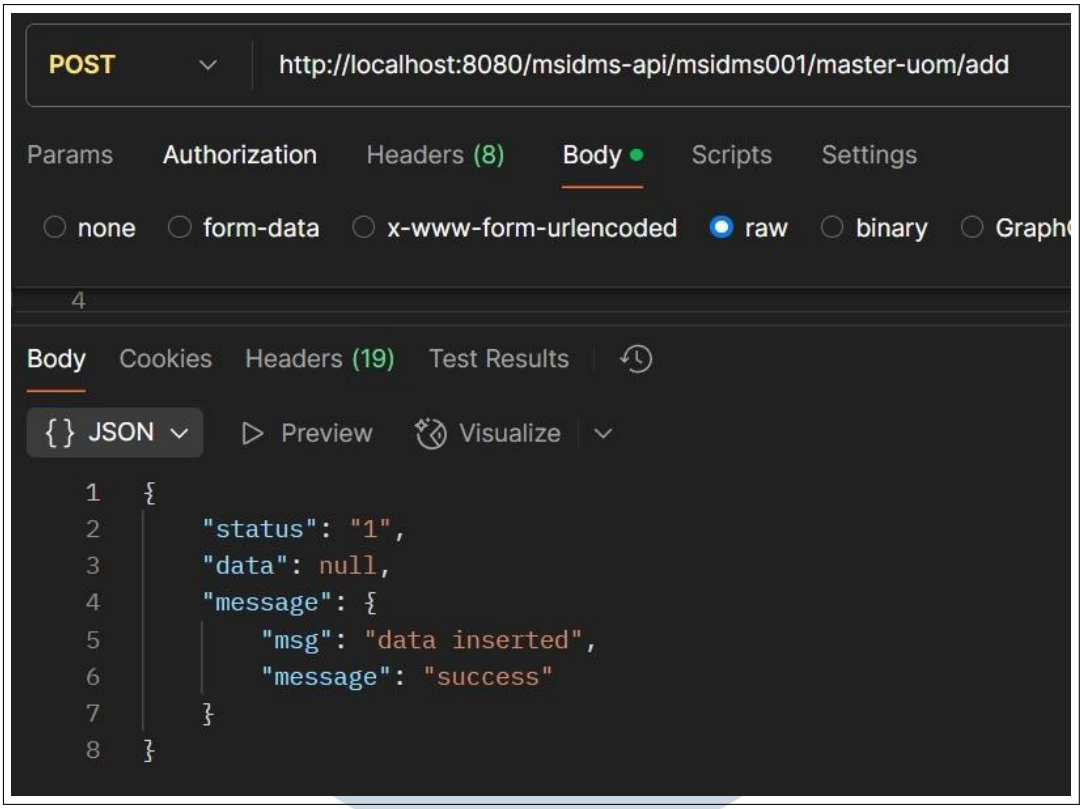
Pada Gambar 3.4, ditampilkan hasil respons dari *endpoint* **/master-uom/** yang diuji menggunakan Postman. Respons yang diterima berupa data berformat JSON, yang berisi informasi lengkap mengenai satuan UoM. Beberapa atribut

penting yang ditampilkan meliputi `id, code, desc, status, created by, modified by` beserta waktunya. Dari pengujian ini, dapat disimpulkan bahwa API berfungsi dengan baik karena mampu menampilkan data secara lengkap dan siap dipanggil oleh *front-end* untuk memanggil seluruh data.



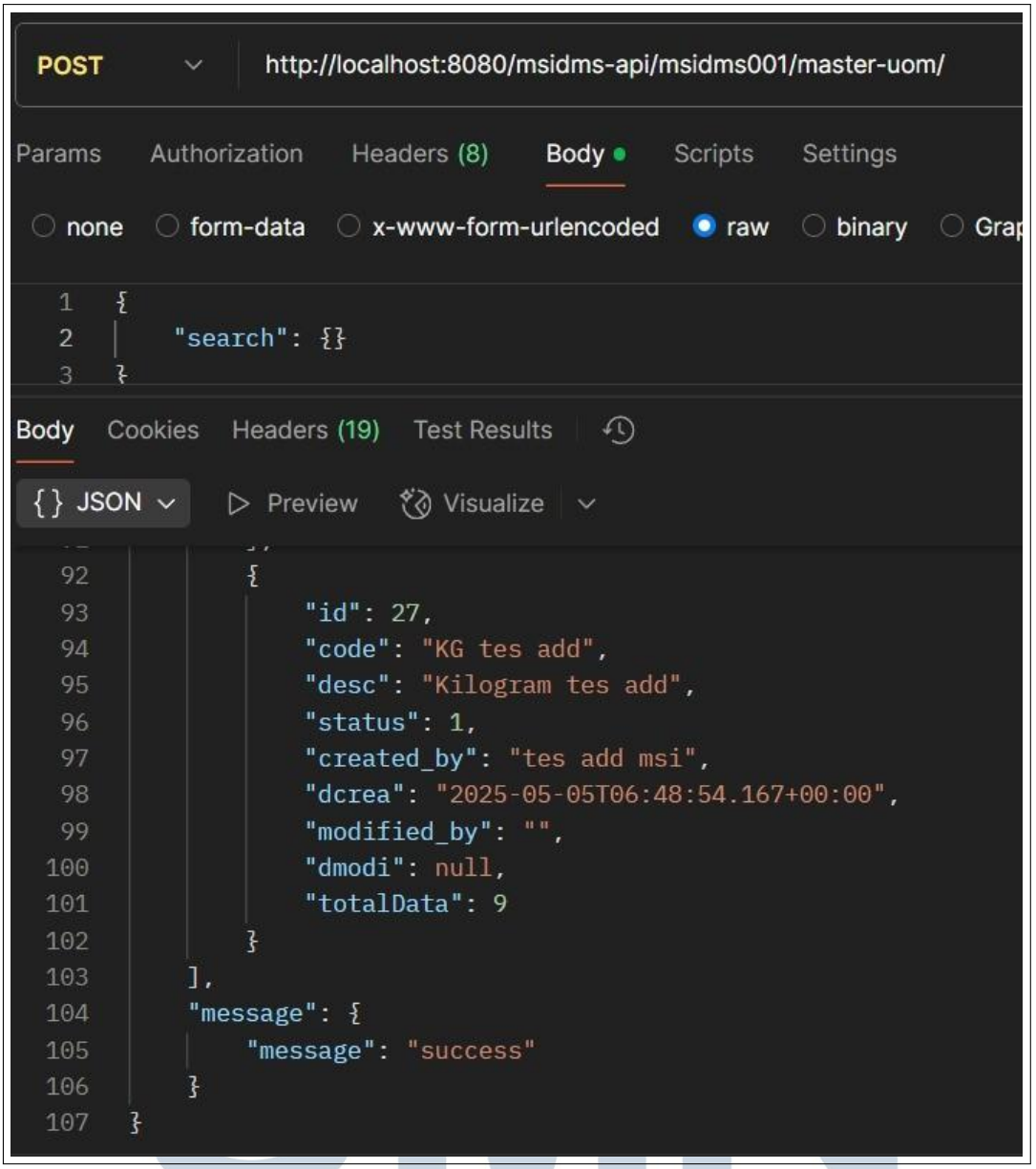
Gambar 3.5. *Add UoM Data* pada Postman

Pada Gambar 3.5, dilakukan pengujian saat ingin menambahkan data UoM. URL yang digunakan untuk menambahkan data adalah **/add** dengan base URL **/master-uom**, sehingga jika digabungkan menjadi **/master-uom/add**. Saat ingin menambah data, pengguna harus mengirim data pada body sehingga *back-end* mampu mengidentifikasi kolom-kolom yang perlu diisi dan ditambahkan menjadi satu data. Pada contoh Gambar 3.5, data yang dimasukkan adalah `code, desc, status, created by` dengan *value* ”KG tes add”, ”Kilogram tes add”,1 , dan ”tes add msi”.



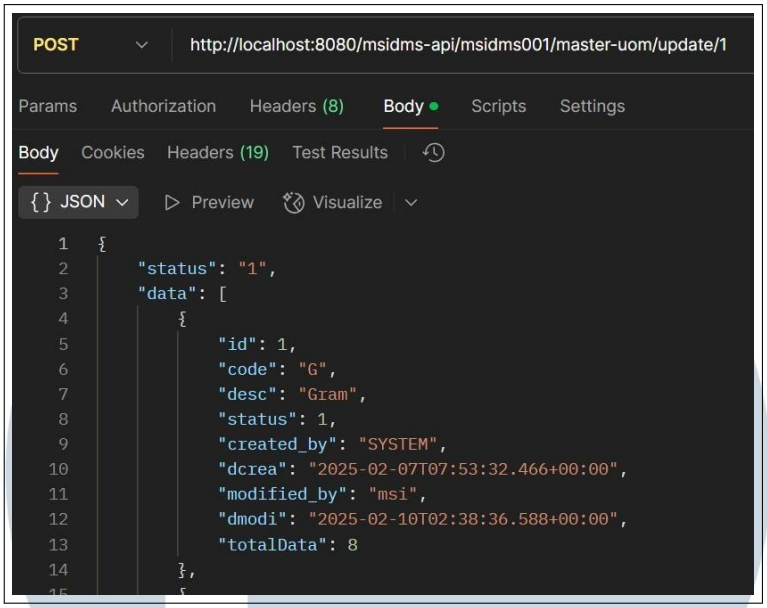
Gambar 3.6. *Success Add UoM Data*

Jika penambahan data berhasil dilakukan, maka akan mengeluarkan pesan berhasil seperti pada Gambar 3.6 dengan keterangan ”*data inserted*” dan pesan berupa ”*success*”. Saat pesan berhasil ditampilkan, langkah selanjutnya adalah melihat apakah data sudah berhasil masuk.



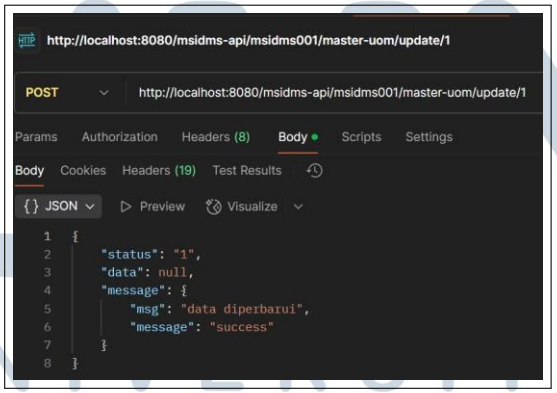
Gambar 3.7. *Check UoM Data*

Selanjutnya dilakukan pengujian untuk memodifikasi data (*update*) pada data UoM melalui Postman. Pengujian ini menggunakan *endpoint* **/update/:updateid** dan metode POST, dimana `:updateid` merupakan ID dari data yang akan diperbarui. Pada contoh kasus ini, proses pembaruan dilakukan pada data dengan ID ke-1.



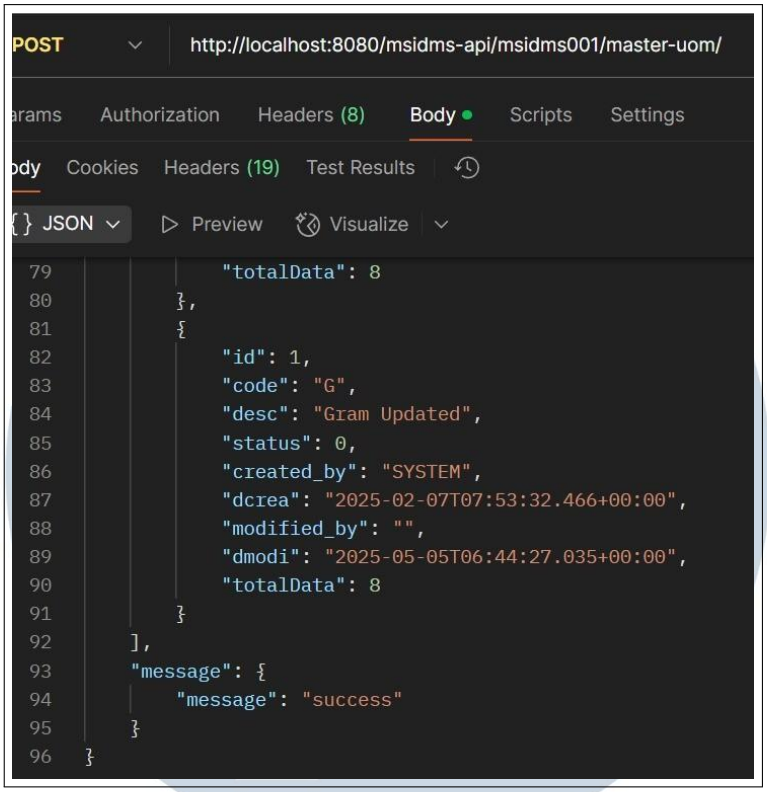
Gambar 3.8. *Update UoM Data*

Dalam melakukan *update* data memiliki konsep yang sama seperti saat menambahkan data. Pada Postman, pengguna harus menuliskan *body* untuk memberitahu *back-end* data mana yang ingin dimodifikasi. Pada contoh ini, digunakan data `desc` dan `status`, yaitu dengan mengganti *value* nya dari ”Gram” menjadi ”Gram Updated” dan status dari 1 menjadi 0 (*inactive*).



Gambar 3.9. *Success Update UoM Data*

Setelah permintaan *update* dikirim, server merespons dengan pesan yang menunjukkan bahwa data berhasil diperbarui, seperti terlihat pada Gambar 3.9. Pada respons tersebut, kolom `desc` telah berubah menjadi ”Gram Updated” dan kolom `status` menjadi 0. Selain itu, nilai `dmodi` atau waktu modifikasi juga diperbarui sesuai dengan waktu eksekusi.



Gambar 3.10. *Check Updated UoM Data*

Perubahan data dipastikan benar-benar tersimpan dengan dilakukan pengambilan ulang data UoM dari server seperti ditampilkan pada Gambar 3.10. Hasil yang ditampilkan menunjukkan bahwa perubahan berhasil dilakukan, dengan nilai-nilai baru yang sesuai seperti yang dikirim pada permintaan sebelumnya.

Fungsi yang terakhir adalah fungsi *delete* yang sesungguhnya tidak benar-benar menghapus data, melainkan hanya mengubah status dari aktif menjadi nonaktif, atau yang biasa disebut dengan *soft-delete*. Fitur menghapus data pada umumnya akan langsung menghapus data pada ID yang diinginkan dengan fungsi `delete()`. Namun, *soft-delete* dilakukan dengan menggunakan API *endpoint* yang sama seperti saat *update* karena *soft-delete* hanya mengubah status.

B Front-end pada Master Unit of Measurement

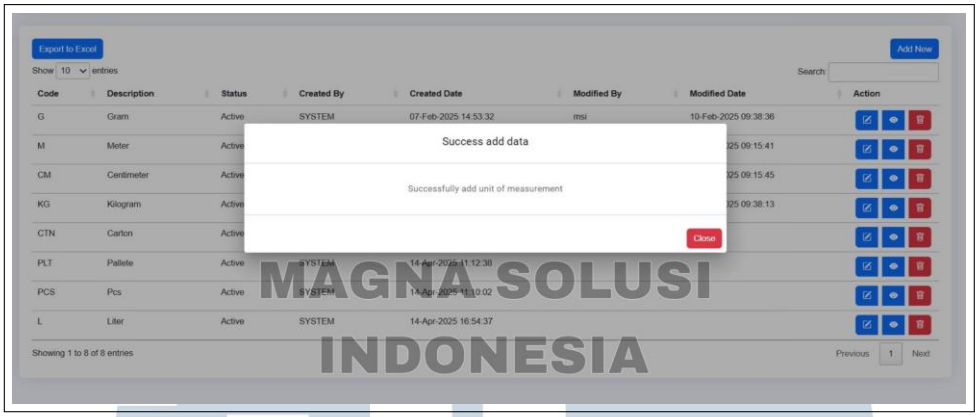
Setelah dilakukan pengujian terhadap API yang dibuat dan semuanya berfungsi dengan baik, maka tahap selanjutnya adalah menjalankan fungsi-fungsi tersebut dengan *front-end*. Tampilan dari *front-end* dan alur kerja aplikasi diuraikan pada penjelasan berikut.

Gambar 3.11. UoM *Add Field*

Gambar 3.11 menunjukkan tampilan saat ingin menambahkan data untuk master UoM. Saat tombol *Add* diklik, akan muncul dialog untuk mengisi data yaitu *code* dan *description*.

Gambar 3.12. UoM *Add Validation*

Pada Gambar 3.12, data yang ditambahkan akan dicek mengikuti validasi yang telah ada, jika memenuhi syarat seperti salah satunya tidak boleh ada *code* yang duplikat, maka *input* berhasil ditambahkan ke *database* dan menampilkan pesan sukses. Jika gagal, maka akan diberikan pesan bagaimana data tidak berhasil ditambahkan, sesuai dengan syarat validasi.



Gambar 3.13. UoM *Add Succeess*

Gambar 3.13 menampilkan jika suatu data berhasil ditambahkan. Data akan dimasukkan ke *database* dengan nid yang bersifat unik untuk mengidentifikasi masing-masing data. Identifikasi dengan nid tersebut sangat bermanfaat saat ingin memodifikasi, melihat, maupun menghapus data. Hal tersebut karena data-data yang dipilih untuk dilakukan aksi-aksi tersebut, akan dibaca melalui nid nya.

Gambar 3.14. UoM *Edit Field*

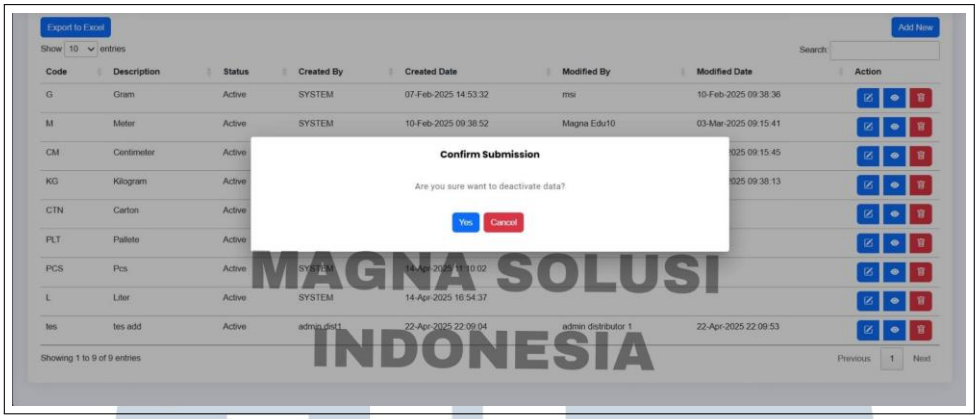
Gambar 3.14 menampilkan dialog saat pengguna ingin memodifikasi data yang telah ada. Sistem menampilkan dialog *form* dengan data yang terisi otomatis. Pada contoh UoM, semua *field* dapat dimodifikasi kecuali *created by, created date, modified by, & modified date*.

Gambar 3.15. UoM *Edit Success*

Gambar 3.15 menampilkan jika data telah berhasil dimodifikasi. Saat pengguna menekan tombol *submit*, maka sistem akan melakukan validasi. Jika validasi terpenuhi, data dikirim ke *back-end* untuk diperbarui, dan pesan sukses akan ditampilkan. Jika tidak valid, sistem menampilkan pesan kesalahan (*error message*).

Gambar 3.16. UoM *View Selected Data*

Terlihat serupa seperti saat ingin memodifikasi data, pada Gambar 3.16 juga terjadi *autofill* pada data yang ingin dilihat. Namun, perbedaan yang dapat dilihat adalah *field-field* setiap data dilakukan *disable* sehingga tidak dapat diubah seperti saat memodifikasi data.



Gambar 3.17. UoM *Delete Confirmation*

Pada Gambar 3.17, saat tombol *delete* ditekan pada suatu data, maka secara tidak langsung pengguna telah memilih suatu data yang ingin dilakukan *soft delete*. Setelah menekan tombol tersebut, sistem akan menampilkan dialog konfirmasi.

Gambar 3.18. UoM *Delete Success*

Setelah proses *soft delete* data berhasil, Gambar 3.18 menampilkan dialog bahwa data berhasil dinonaktifkan. Status yang aktif akan memiliki *value* sebagai 1, sedangkan saat nonaktif, maka *value* nya menjadi 0. Permintaan tersebut akan dikirim ke *back-end* dan menampilkan pesan sukses.

Dengan alur yang diberikan, tampilan dari master UoM untuk proses CRUD dapat dilihat pada Gambar 3.44 hingga Gambar 3.18. Setiap *dashboard* dalam proyek ini umumnya akan memiliki fitur serupa seperti *search, filter*, dan CRUD. Perbedaan utama akan terletak pada data-data yang dimiliki, data yang ditampilkan, data yang dicari, data yang akan ditambah, dimodifikasi, serta hubungannya dengan data lain yang saling berkaitan secara alur bisnis.

C List of Values pada Unit of Measurement Conversion

Setelah membuat master UoM, penugasan dilanjutkan dengan membuat UoM *Conversion* seperti yang telah dijelaskan di awal. Alur kerjanya sebagian besar sama, namun ada penambahan untuk fitur *List of Values* (LoV), yaitu seperti drop-down yang berbentuk *table* untuk menampilkan data yang ingin digunakan untuk keperluan suatu *dashboard*.

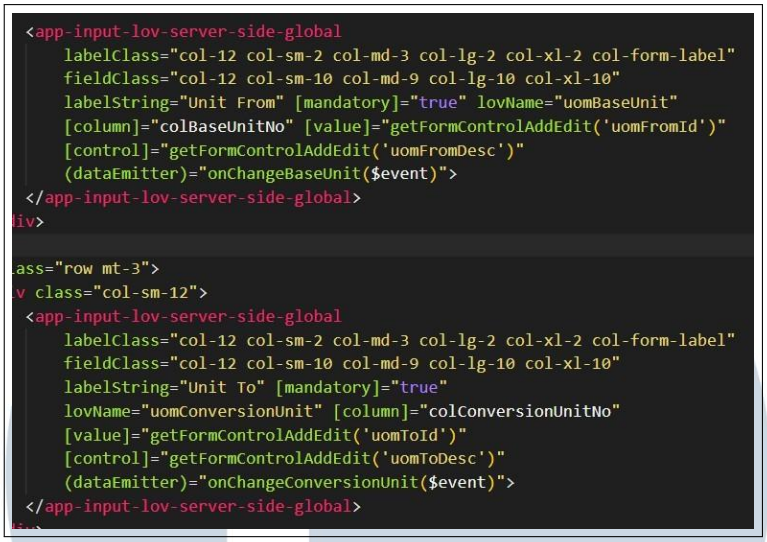
Komponen *List of Values* adalah komponen global yang telah dibuat oleh senior developer pada perusahaan. Namun, setiap developer yang menghubungkan LoV dengan *master dashboard* yang sedang dikerjakan, perlu menyesuaikan dan mengintegrasikan LoV tersebut secara mandiri. Oleh karena itu, LoV yang sudah ada akan diintegrasikan dengan master UoM yang sedang dikerjakan.

List of Values menginisialisasi kolom dan data LoV untuk menyiapkan wadah bagi data-data yang ingin dipanggil dan dimunculkan. Sistem memanggil API ke tabel UoM menggunakan metode POST. Jika respons sesuai, sistem akan melakukan pengecekan nama LoV dan memetakan data yang sesuai, sebelum menampilkan daftar data UoM. Rincian dari pemanggilan API UoM dapat dilihat pada Tabel 3.3.

Tabel 3.3. Tabel LoV UoM *Endpoint*

Nama API	**Endpoint**	**Metode**	**Deskripsi**
Get All UoM	/	POST	API ini berfungsi untuk mendapatkan semua data UoM agar dapat ditampilkan pada tabel LoV.

Pengambilan data UoM pada halaman master UoM dan data *List of Values* di UoM Conversion menggunakan API yang sama yaitu *endpoint* **/master-uom/** dengan metode POST. Perbedaannya hanya terletak pada tempat penggunaannya, di mana satu digunakan untuk menampilkan data utama UoM, dan satu lagi sebagai referensi pilihan pada fitur konversi UoM.

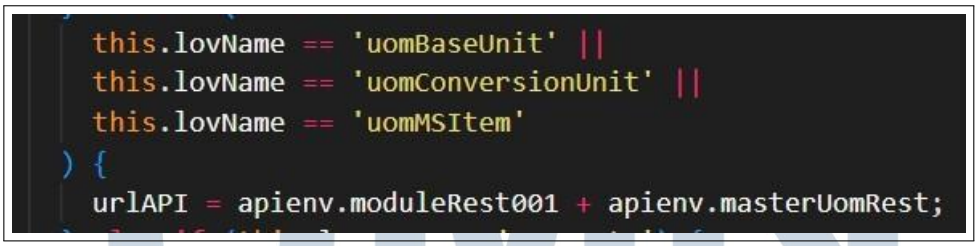


```
<app-input-lov-server-side-global
    labelClass="col-12 col-sm-2 col-md-3 col-lg-2 col-xl-2 col-form-label"
    fieldClass="col-12 col-sm-10 col-md-9 col-lg-10 col-xl-10"
    labelString="Unit From" [mandatory]="true" lovName="uomBaseUnit"
    [column]="colBaseUnitNo" [value]="getFormControlAddEdit('uomFromId')"
    [control]="getFormControlAddEdit('uomFromDesc')"
    (dataEmitter)="onChangeBaseUnit($event)">
</app-input-lov-server-side-global>
iv>

ass="row mt-3">
v class="col-sm-12">
<app-input-lov-server-side-global
    labelClass="col-12 col-sm-2 col-md-3 col-lg-2 col-xl-2 col-form-label"
    fieldClass="col-12 col-sm-10 col-md-9 col-lg-10 col-xl-10"
    labelString="Unit To" [mandatory]="true"
    lovName="uomConversionUnit" [column]="colConversionUnitNo"
    [value]="getFormControlAddEdit('uomToId')"
    [control]="getFormControlAddEdit('uomToDesc')"
    (dataEmitter)="onChangeConversionUnit($event)">
</app-input-lov-server-side-global>
```

Gambar 3.19. Pemanggilan LoV UoM dari HTML pada UoM Conversion

Pada Gambar 3.19, sistem melakukan proses pemanggilan API yang melibatkan sisi *front-end* dan *back-end*. Proses ini dimulai ketika komponen `app-input-lov-server-side-global` diinisialisasi. Komponen ini memuat properti `lovName, column`, dan `control`, yang menentukan jenis data LoV yang akan ditampilkan (misalnya, `uomBaseUnit` atau `uomConversionUnit`.



```
    this.lovName == 'uomBaseUnit' ||
    this.lovName == 'uomConversionUnit' ||
    this.lovName == 'uomMSItem'
) {
    urlAPI = apienv.moduleRest001 + apienv.masterUomRest;
```

Gambar 3.20. Pemanggilan API UoM untuk Digunakan oleh UoM Conversion

Gambar 3.20 menunjukkan fungsi `initDataLOV()` dipanggil untuk membentuk struktur *DataTable* dan mendefinisikan metode *ajax* yang digunakan untuk memuat data dari server. Pada dasarnya bagian ini adalah penghubung untuk memanggil API data UoM dari LoV UoM Conversion.

Gambar 3.21. *Mapping* Data LoV

Setelah data diterima dari API, fungsi `getDataFromAPI()` akan memeriksa validitas respon seperti pada Gambar 3.21. Bila data yang diterima tidak kosong, sistem akan melakukan penyaringan berdasarkan status aktif (1) dan memetakan ulang data berdasarkan nama LoV yang dipilih. Contohnya jika nama LoV adalah uomBaseUnit, maka data dari API akan dipetakan menjadi uomFromCode dan uomFromDesc. Hasil pemetaan tersebut akan ditampilkan dalam tabel LoV untuk dipilih oleh *user*.

```
sql += " SELECT nid, vcode, vdesc, nstatus, vcrea, dcrea, vmodi, dmodi ";
sqlTable += " FROM tblm_uom ";
sqlWhere += " WHERE 1=1 ";
```

Gambar 3.22. *Query Get* Data UoM

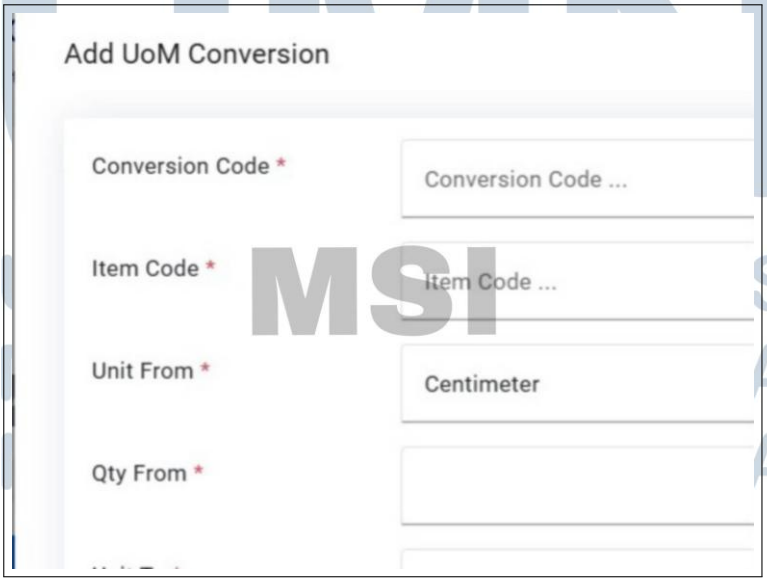
Gambar 3.22 menunjukkan sisi *back-end* yang dikirim dari *front-end*. Permintaan tersebut ditangani oleh fungsi `getData()` pada kelas `MasterUoMDaoImpl`. Fungsi ini membentuk *SQL query* yang akan menampilkan semua data yang berada di dalam `SELECT` *statement* tersebut. Meskipun telah menggunakan LoV *server-side*, namun kegunaannya tidak terasa jika jumlah

data hanya sedikit seperti UoM. Penggunaan *server-side* dalam LoV akan lebih dijelaskan dalam Master Location.

Setelah melewati beberapa proses untuk memanggil LoV tersebut, visualisasi dari penggunaan LoV akan lebih jelas dengan memperhatikan beberapa dokumentasi aplikasi dari Gambar 3.23 hingga 3.25.

Gambar 3.23. UoM *Conversion* LoV Display

Pada Gambar 3.23, terjadi pemanggilan API ke tabel UoM dan pemetaan data yang sesuai, lalu menampilkan data tersebut. Setelah menampilkan semua data UoM dan memilih salah satu data LoV, sistem akan menyimpan ID dari data yang dipilih sebagai nilai yang sebenarnya (yang akan dikirim ke *database* sebagai *foreign key*).



Gambar 3.24. UoM *Conversion* Selected LoV

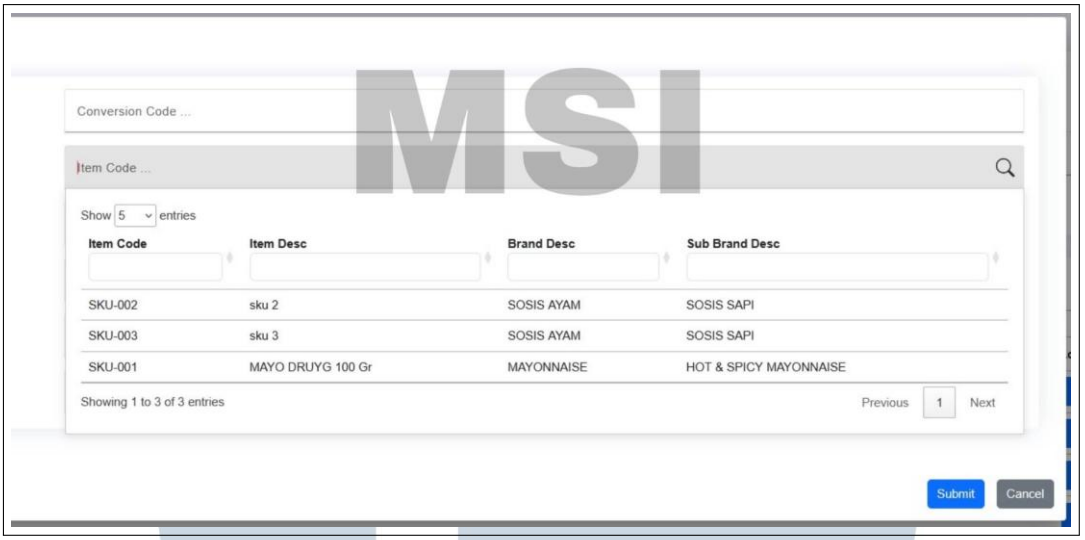
Pada Gambar 3.24, sistem menampilkan nilai lain yang lebih deskriptif, seperti nama atau keterangan dari ID data terpilih agar lebih mudah dipahami oleh pengguna. Contoh, jika ID yang tersimpan adalah angka 3, maka *value* yang akan ditampilkan adalah deskripsi atau nama dari ID tersebut, misalnya centimeter, meter, kilogram, dan sebagainya.

Gambar 3.25. UoM *Conversion* LoV Display

Gambar 3.25 memiliki konsep yang sama seperti sebelumnya, hanya saja perbedaannya terletak pada konteks yang digunakan, di mana LoV ini digunakan untuk memilih tujuan satuan konversi (*Unit To*), sedangkan LoV sebelumnya digunakan untuk memilih asal satuan konversi (*Unit From*).

D List of Values dengan Kustom Query untuk Menampilkan Data Item pada Unit of Measurement Conversion

Selain *Unit To* dan *Unit From*, terdapat LoV untuk memilih data barang (item) saat menambah maupun memodifikasi data. Pada LoV yang dikembangkan, proses pemilihan data barang tidak hanya terbatas pada menampilkan SKU (*Stock Keeping Unit*) saja, melainkan juga menampilkan informasi terkait sub-brand dan brand dari setiap SKU tersebut. Kebutuhan ini muncul karena dalam proses bisnis, sering kali diperlukan pemahaman hierarki mengenai suatu barang, mulai dari brand induk, sub-brand, hingga SKU paling akhir. Oleh karena itu, *List of Values* (LoV) untuk item diimplementasikan dengan teknik `SELF JOIN` pada tabel item, sehingga mampu menampilkan hubungan antar tingkatan item secara otomatis dalam satu tampilan.



Gambar 3.26. *Choose Item Data*

Gambar 3.26 menampilkan data barang pada aplikasi. Pada tampilan ini, setiap baris tidak hanya menampilkan informasi utama dari SKU seperti kode barang dan deskripsinya, tetapi juga menampilkan dua kolom tambahan yaitu **Brand Desc** dan **Sub Brand Desc**. Dengan struktur ini, user dapat langsung melihat secara hierarkis dari level tertinggi (brand), menengah (sub-brand), hingga level terendah (SKU).

Dalam upaya mencapai hasil yang diinginkan, pembuatan *query* SQL untuk menampilkan data detail dari SKU, melibatkan arahan dan diskusi dengan seorang senior developer. Diskusi yang dilakukan untuk melakukan `SELF JOIN` pada tabel item menjadi kunci untuk membentuk hierarki data dari brand, sub-brand, hingga ke level SKU.

```
CREATE or replace VIEW vw_item_sku_with_brand_subbrand AS
SELECT
    brand.vitem_desc AS vbrand_desc,
    subbrand.vitem_desc AS vsubbrand_desc,
    sku.*
FROM tblm_item sku
INNER JOIN tblm_setting setting
    ON sku.vitem_type = setting.vcode AND setting.vtype = 'item_type'
LEFT JOIN (
    SELECT a.*
    FROM tblm_item a
    INNER JOIN tblm_setting sb
        ON a.vitem_type = sb.vcode AND sb.vtype = 'item_type'
    WHERE sb.vcode = '2'
) AS subbrand
    ON sku.nparent_id = subbrand.nid
LEFT JOIN (
    SELECT a.*
    FROM tblm_item a
    INNER JOIN tblm_setting br
        ON a.vitem_type = br.vcode AND br.vtype = 'item_type'
    WHERE br.vcode = '1'
) AS brand
    ON subbrand.nparent_id = brand.nid
WHERE setting.vcode = '3';
```

Gambar 3.27. *Query to Show Item Data*

Gambar 3.27 memperlihatkan struktur *query* SQL yang digunakan untuk membangun data LoV item. Di sini digunakan `self-join` secara berjenjang pada tabel item. Query pada level SKU mengambil data dari tabel item yang bertipe SKU (vitem_type = 3). Setelah itu, setiap baris SKU di-join-kan ke parent-nya (sub brand) dengan mencocokkan field `nparent_id` (pada SKU) ke nid (pada sub-brand) dan memastikan parent bertipe sub-brand (vitem_type = 2). Kemudian, dari hasil join sebelumnya, sub-brand akan kembali di-join-kan ke parent berikutnya (brand) menggunakan field yang sama, dan hanya memilih parent bertipe brand (vitem_type = 1).

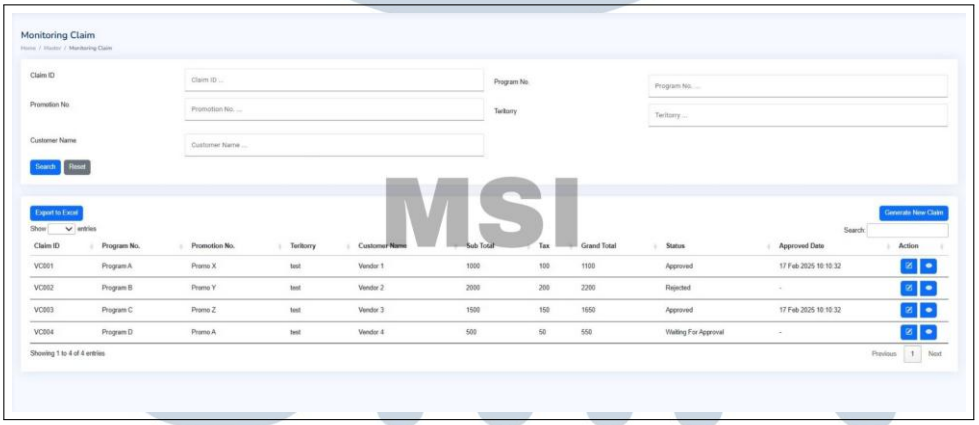
```
sql += "SELECT * ";
sqlTable += " FROM vw_item_sku_with_brand_subbrand ";
sqlWhere += " WHERE 1=1 ";
```

Gambar 3.28. *Query to Call View*

Gambar 3.28 menunjukkan contoh implementasi kode program pada aplikasi untuk memanggil data item dari *database*. *View* tersebut adalah hasil dari *query* yang sudah dijelaskan pada gambar sebelumnya. Dengan cara ini, aplikasi hanya perlu melakukan satu kali pengambilan data, tanpa perlu melakukan *join* ulang atau pemrosesan tambahan di sisi aplikasi. Selain itu, hal ini juga memudahkan developer lain untuk langsung menggunakan *view* yang telah tersedia.

3.3.3 Membuat Front-End dari Monitoring Claim

Monitoring claim adalah fitur yang digunakan distributor untuk melacak, mencatat, dan mengajukan klaim biaya promosi atau *rebate* dari vendor atau pabrik (*principal*). Tujuan dari *dashboard* ini digunakan untuk melihat daftar semua klaim yang pernah dibuat, melihat *invoice* yang menjadi dasar klaim, dan pengajuan klaim baru ke vendor atau promosi yang telah dijalankan. Tampilan *monitoring claim* dimulai dengan menampilkan *form* pencarian dan tombol-tombol yang biasanya digunakan dalam *dashboard*, lalu menampilkan tabel berisi data *dummy/hard coded*, seperti yang terlihat pada Gambar 3.29.



Gambar 3.29. *Monitoring Claim Display*

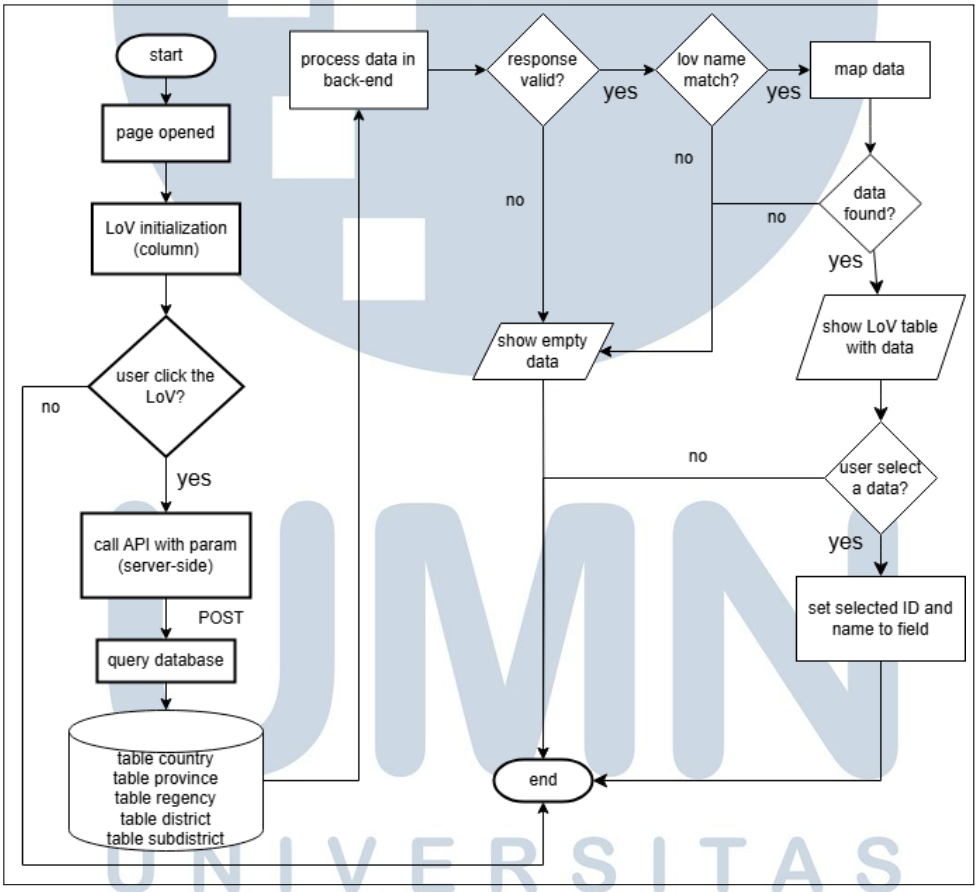
Gambar 3.29 menampilkan *dashboard monitoring* milik *monitoring claim*. Sama seperti *dashboard* lainnya, pada *monitoring claim* juga nantinya dapat melakukan *searching* terhadap data-data yang ada di dalam tabel. Kata kunci yang digunakan untuk mencari data tersebut berupa *Claim ID, Promotion Number, Customer Nme, Program Number*, dan *Territory*. Selain itu, juga terdapat tombol *reset* untuk memunculkan semua data kembali tanpa menggunakan *filter search*.

3.3.4 Perbaikan Fitur-Fitur pada Master Location

Tugas selanjutnya adalah untuk memperbaiki Master Location yang rusak akibat dari penyesuaian fitur *List of Values* yang baru, selain itu juga terdapat beberapa tambahan validasi seperti input data dan *mandatory field*.

A LoV pada Master Location

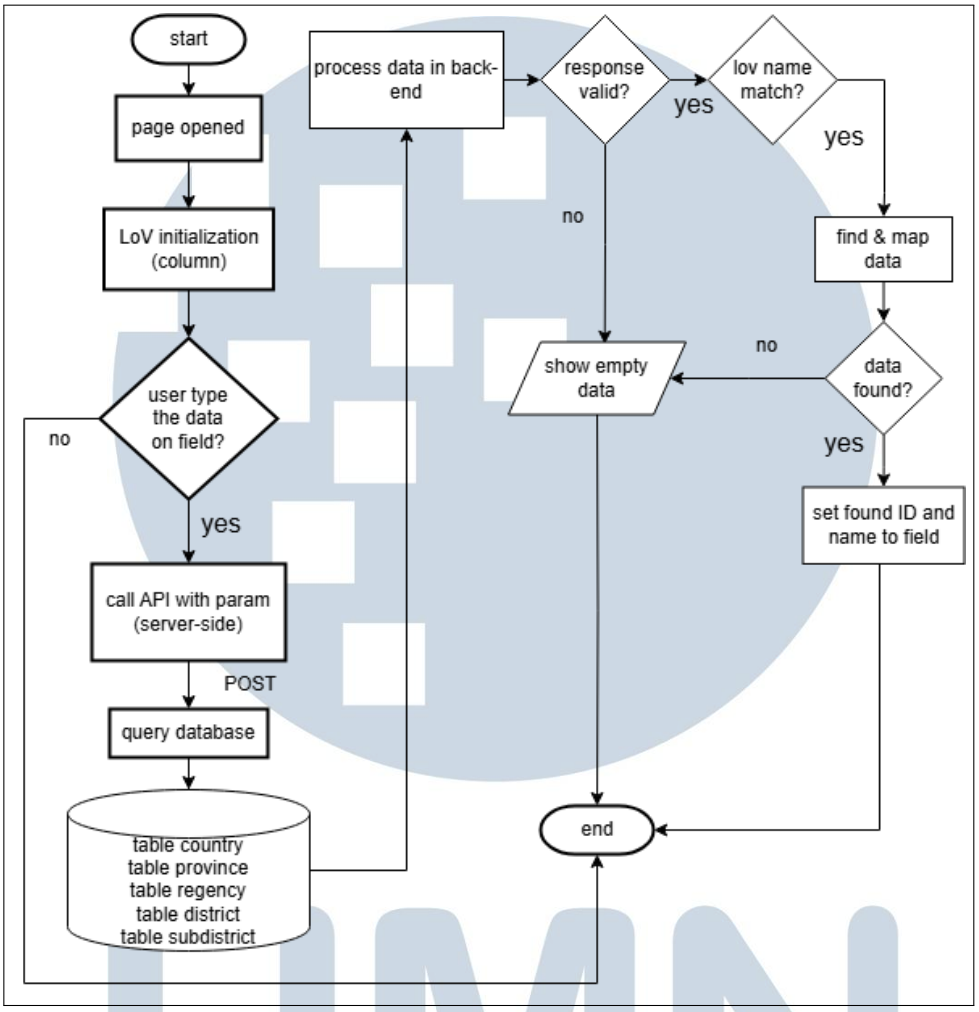
Pemanggilan data pada LoV dibagi menjadi 2, yaitu LoV saat *click* dan LoV saat *type*. Perbedaan tersebut akan dijelaskan dalam *flowchart* berikut.



Gambar 3.30. Flowchart LoV saat Click

Gambar 3.30 menjelaskan LoV saat **click**. LoV yang dibuka dengan cara *click* akan langsung memanggil data ke *back-end* untuk menampilkan tabel dari data tertuju. Setelah API nya cocok, maka akan langsung menampilkan data dalam bentuk tabel LoV, jika data tidak ditemukan maka akan menampilkan tabel dengan tulisan ”*No Data Available on Table*”. Saat kumpulan data telah ditampilkan, *user*

dapat memilih salah satu data untuk diset ke *field* input.

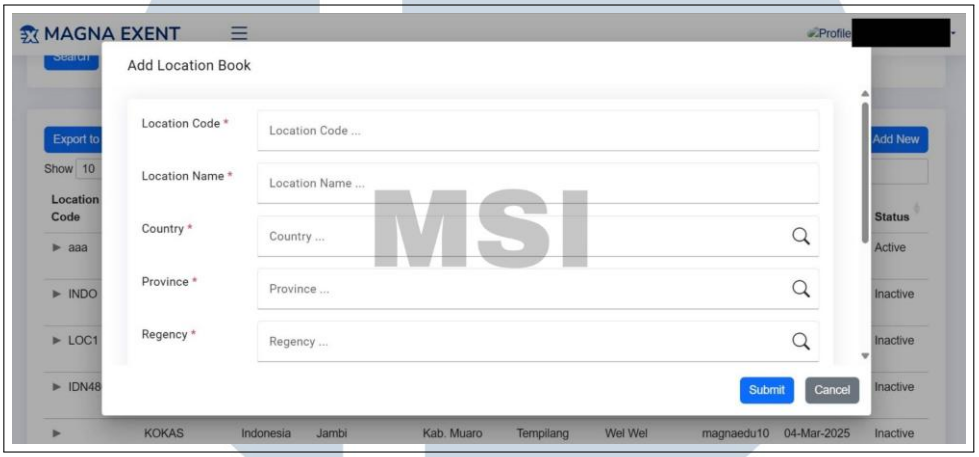


Gambar 3.31. Flowchart LoV saat Type

Gambar 3.31 menjelaskan LoV saat **type**. LoV yang dilakukan dengan cara mengetik langsung pada *field* input, setiap karakter yang diketik, akan dikirim langsung dalam bentuk parameter ke *back-end*. Sehingga, setiap karakter dicek apakah data yang ingin dipilih ada dalam *database*. Jika ada, data dapat langsung diset ke *field*, namun jika tidak ada maka akan menampilkan "*No Data Available on Table*".

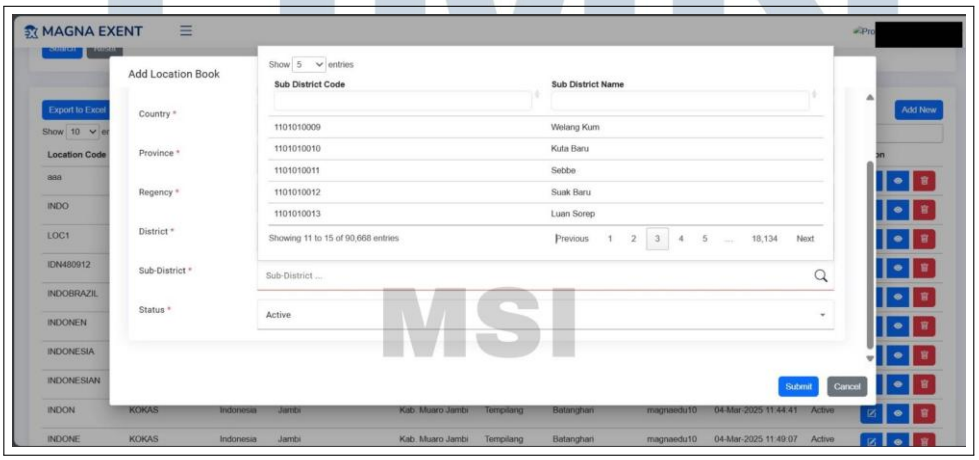
Master location menyimpan data lokasi lengkap beserta *foreign key* dari tabel negara, provinsi, kabupaten/kota, kecamatan, dan kelurahan untuk menampilkan data lokasi secara lengkap. Jumlah data yang sangat besar yaitu lebih dari 90.000, menyebabkan penggunaan LoV biasa membuat tampilan menjadi

lambat. Oleh karena itu, dilakukan pembuatan ulang LoV berbasis *server-side* oleh seorang senior developer agar proses pencarian dan penampilan data lebih cepat dan efisien. Penerapan LoV baru yang berlaku menyebabkan beberapa master menjadi *error*, sehingga dibutuhkan penyesuaian ulang pada master-master tersebut, salah satunya adalah master *location*. Pada contoh ini, digunakan LoV dari *sub-district* sebagai perwakilan dari data lokasi lain.



Gambar 3.32. *Location LoVs*

Gambar 3.32 menunjukkan tampilan saat membuka *field*. Di tahap ini, kolom LoV telah diinisialisasi. Hal tersebut karena *function* untuk memanggil LoV diletakkan di awal program. Kolom tersebut akan tetap dimunculkan saat LoV dibuka, meskipun jika tidak ada data yang ditemukan. Jika tidak ada data yang ditemukan, maka akan mengembalikan tulisan *No data available in table*.



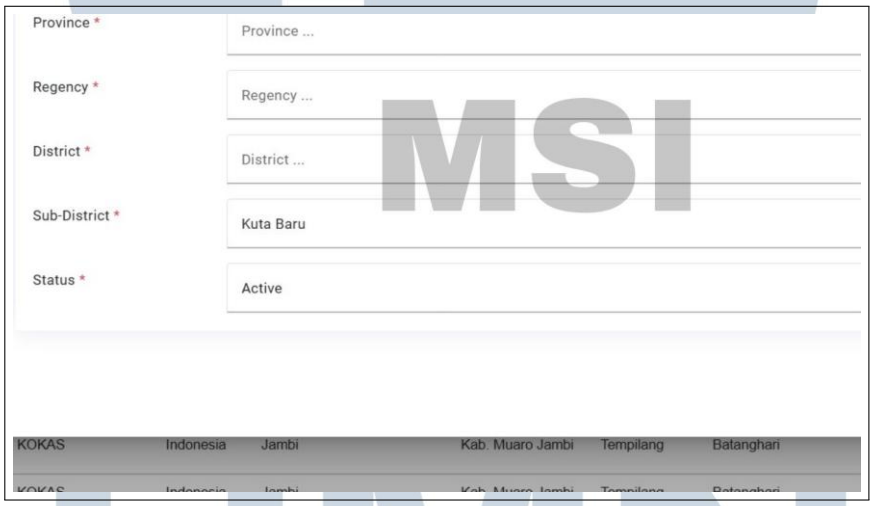
Gambar 3.33. *Location LoV on Click Image*

Gambar 3.33 menunjukkan tampilan ketika *user* mengklik LoV untuk

menampilkan seluruh data dan akan memilih data, dalam hal ini contoh data yang akan dipilih adalah ”Kuta Baru”. *Sub-district* dipilih karena jumlah datanya sangat banyak, sehingga penerapan *server-side* LoV menjadi sangat krusial.

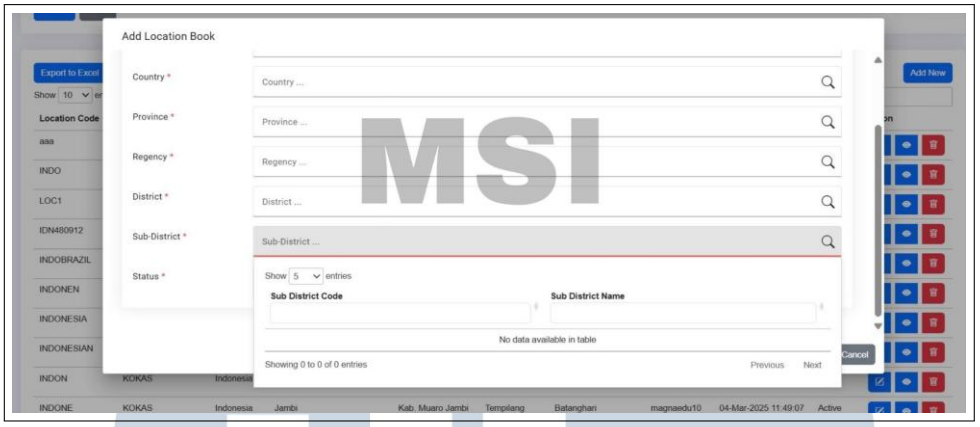
Tanpa *server-side*, seluruh data akan diproses dan dimuat sekaligus saat LoV dibuka, yang dapat menyebabkan performa lambat. Sebaliknya dengan *server-side*, data diproses secara bertahap.

Front-end mengirim permintaan ke *back-end* dengan parameter seperti *offset, limit, sort, order*, dan *filter*. Pada contoh Gambar 3.33, hanya 5 data yang ditampilkan per halaman (*limit = 5*) dan *offset* dimulai dari 0. Saat pengguna membuka halaman berikutnya, *offset* akan meningkat, sementara *limit* tetap, sehingga proses tetap efisien tanpa membebani sisi klien.



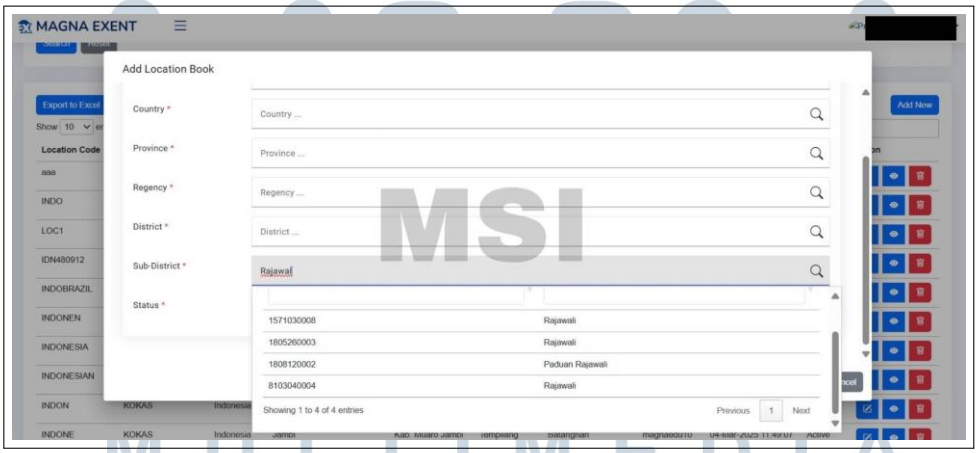
Gambar 3.34. *Location LoV on Click Image*

Setelah pengguna memilih salah satu data tabel LoV, dalam contoh ini adalah *sub-district* ”Kuta Baru”, maka sistem akan mengambil data ID dan nama dari entri tersebut, lalu menyimpannya ke dalam *form field* terkait. Proses ini dilakukan melalui fungsi *setFieldLOV*, yang mengatur nilai ID tersembunyi (biasanya digunakan untuk pemrosesan di *back-end*) dan menampilkan nama lokasi secara visual di kolom input seperti ditunjukkan pada Gambar 3.34. Dengan pendekatan ini, pengguna tidak perlu mengetahui ID sebenarnya dari data terpilih, karena sistem secara otomatis menanganinya di balik layar.



Gambar 3.35. *Location LoV Not Found*

Kondisi seperti yang ditunjukkan pada Gambar 3.35 terjadi apabila pengguna melakukan pencarian namun tidak ada data sesuai yang ditemukan oleh sistem. Hal ini bisa terjadi karena data yang dicari memang tidak tersedia pada *database*, atau pengguna salah mengetikkan kata kunci. Dalam LoV *server-side*, sistem menampilkan tabel kosong beserta keterangan "*No data available in table*" jika hasil *query* dengan parameter tertentu (seperti *filter, offset*, dan *limit*) tidak mengembalikan data apa pun. Tampilan ini memberikan informasi bahwa meskipun tidak menemukan hasil yang relevan, tetapi permintaan dari *front-end* berhasil diproses hingga *back-end*.



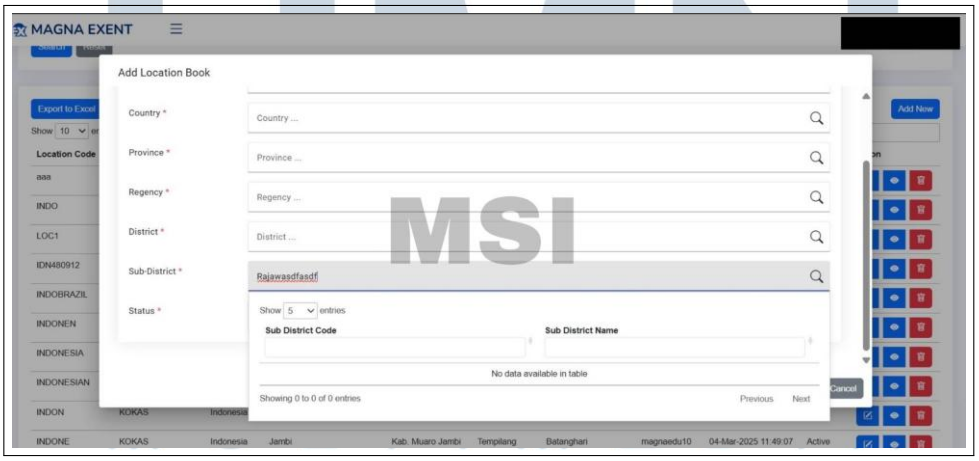
Gambar 3.36. *Location LoV on Type Image*

Pada Gambar 3.37, pengguna mengetik langsung pada *field sub-district* untuk mencari data tertentu. Saat sedang mengetik, maka fungsi *setFilterTyping()* akan dipersiapkan dengan kata kunci yang diatur (pada contoh ini adalah nama lokasi, bukan kode). Nilai dari kata kunci tersebut akan digunakan oleh fungsi

initDataLOV() untuk menginisialisasi dan menampilkan data LoV. Pada proses ini, *DataTables* akan mengirimkan permintaan ke server menggunakan parameter seperti *offset, limit, sort, order*, serta *filter* berdasarkan *input* pengguna.

Gambar 3.37. *Location LoV on Type Image*

Pada Gambar 3.37, data yang diketik akan disimpan ke dalam *field sub-district*, dengan syarat bahwa data tersebut harus ada. Jika terdapat beberapa pilihan dengan nama yang sama, maka akan diambil data yang paling atas.



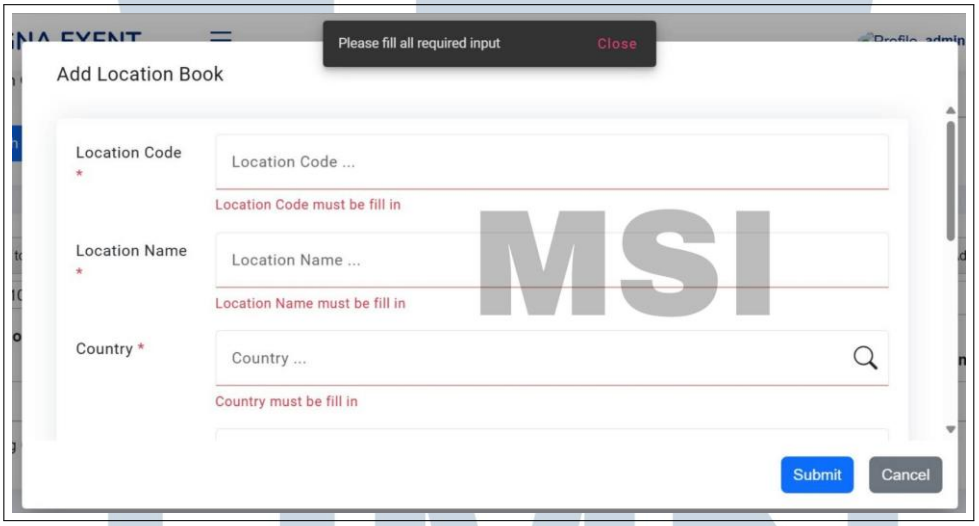
Gambar 3.38. *Location LoV Not Found*

Gambar 3.38 menunjukkan kondisi ketika pengguna mengetik kata kunci pada *field sub-district*, namun data yang dimaksud tidak tersedia di dalam *database*. Dalam kasus ini, permintaan tetap dikirim ke server dengan parameter pencarian

melalui mekanisme *server-side*. Server memproses filter tersebut, dan jika hasil pencarian tidak ditemukan kecocokan, maka tabel pada LoV akan dikembalikan dalam kondisi kosong.

B Menerapkan Mandatory Field pada Master Location

Pada Master Location, beberapa *input* menggunakan properti bernama *mandatory* untuk menandai apakah suatu *field* wajib diisi atau tidak. Secara umum, *mandatory field* adalah komponen *input* yang tidak boleh dikosongkan oleh pengguna. Tujuan dari penggunaannya adalah untuk memastikan bahwa data yang dianggap penting telah diisi benar oleh pengguna.



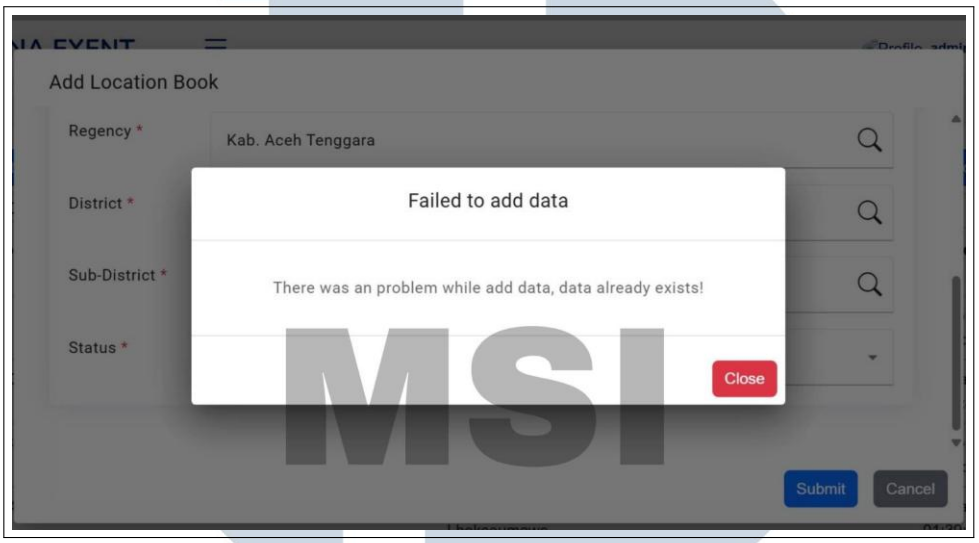
Gambar 3.39. *Mandatory Field* bernilai *TRUE*

Pada Gambar 3.39, ketika properti diset ke nilai *TRUE*, maka akan ditampilkan tanda bintang merah (*) di sebelah *label field*, menandakan bahwa *field* tersebut wajib diisi. Jika ada salah satu saja *field* dengan nilai *TRUE* yang tidak diisi, maka validasi akan gagal dan *submit* data tidak berhasil dengan pesan kesalahan. Sebaliknya, jika *mandatory* bernilai *FALSE*, maka pengguna diperbolehkan untuk melewati *field* tersebut tanpa menyebabkan kegagalan validasi.

C Input Field Bersifat Unik

Dalam proses penambahan dan pembaruan data lokasi pada Master Location, diterapkan validasi khusus untuk memastikan bahwa data yang dimasukkan bersifat unik dan tidak duplikat. Validasi ini memeriksa apakah nilai

`vcode` dan `vname` sudah pernah digunakan sebelumnya. Jika terdapat data lain dengan kombinasi `vcode` atau `vname` yang sama, maka sistem akan menghentikan proses simpan atau modifikasi, dan menampilkan pesan kesalahan seperti pada Gambar 3.40.

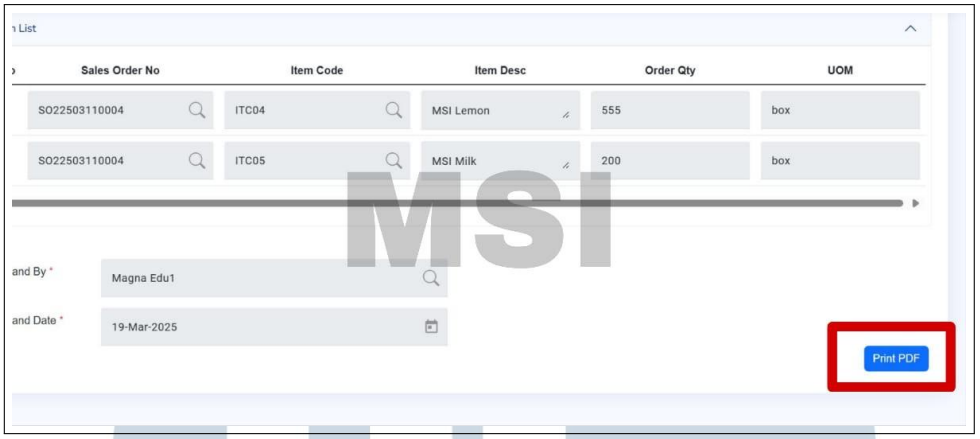


Gambar 3.40. Validasi Input bersifat Unik

Apabila tidak ditemukan data yang sama, maka proses simpan (*insert*) maupun modifikasi (*update*) akan dilanjutkan. Selain itu, pada saat *update*, sistem juga memverifikasi apakah perubahan yang dilakukan memang benar-benar berbeda dari data sebelumnya, sehingga pembaruan yang tidak perlu dapat dihindari. Validasi ini dilakukan dengan melakukan pengecekan terhadap semua kombinasi *input field*, jika kombinasi `vcode, vname, country, province, regency, district`, dan `sub-district` bernilai sama, maka validasi akan gagal dan tidak dapat dilanjutkan. Ketentuan ini berlaku bagi setiap master dashboard yang ada.

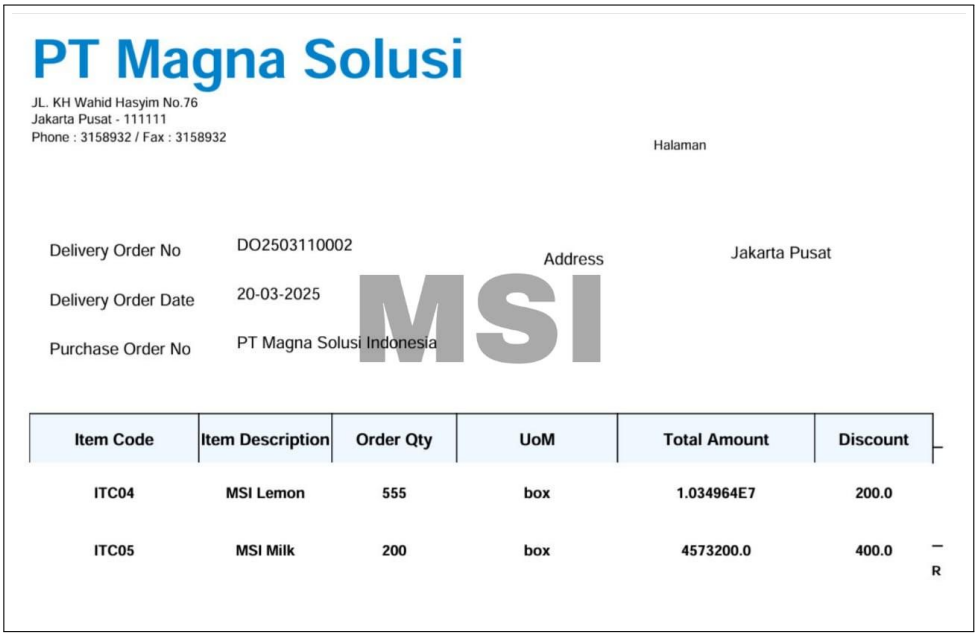
3.3.5 Mengintegrasikan Jaspersoft Studio dengan Aplikasi untuk Mencetak Report dalam Bentuk PDF

Pada halaman *Delivery Order*, pengguna dapat melihat detail informasi pesanan yang telah tersimpan, seperti nama pengemudi, nomor *sales order*, catatan, daftar *item*, serta informasi pihak pemesan. Halaman ini bersifat *read-only*, sehingga data tidak dapat diedit.



Gambar 3.41. Tombol Cetak PDF pada Delivery Order

Gambar 3.41 menunjukkan di bagian kanan bawah terdapat tombol **"Print PDF"** yang berfungsi untuk mencetak dokumen format PDF. Ketika tombol ditekan, sistem menjalankan proses otomatis yang mengambil seluruh data dari tampilan tersebut, kemudian diolah melalui *template* laporan menggunakan **JasperReports**. *Template* ini telah didesain sebelumnya, sehingga hanya dilakukan *mapping* terhadap data-data yang masuk dan mengeluarkannya dalam *reports* tersebut. Proses ini dilakukan melalui *endpoint back-end* dengan ID tertentu agar dapat mengidentifikasi data detail dari ID tersebut.



Gambar 3.42. Bentuk Template Report dalam PDF

Pada Gambar 3.42, hasilnya akan berupa file PDF yang menampilkan

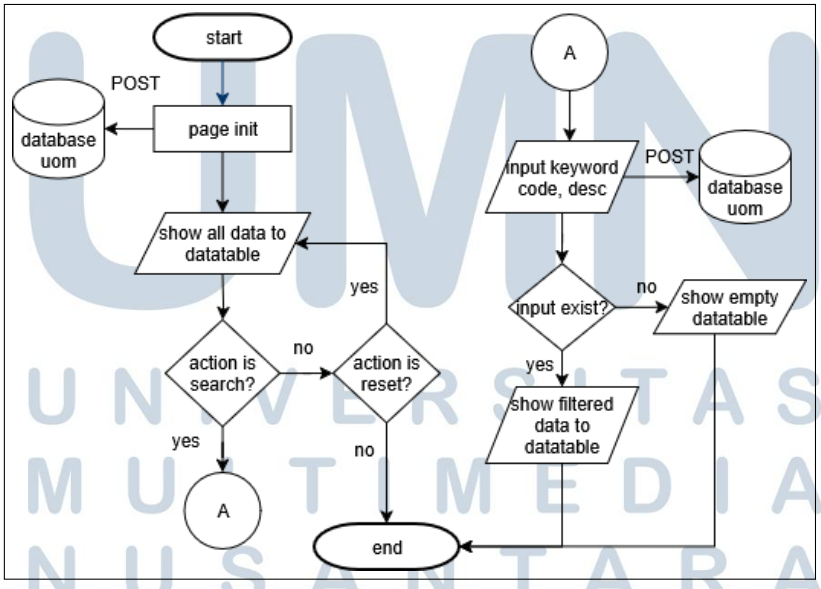
seluruh data dengan format yang telah ditentukan, dengan informasi seperti nomor dan tanggal *Delivery Order*, nama distributor dan alamat, daftar *item* pesanan dengan jumlah, satuan, total, dan diskon, serta ruang tanda tangan untuk pihak terkait. Dengan format tersebut, pencetakan dokumen dapat dipermudah dan menjamin konsistensi format laporan.

3.3.6 Menerapkan Filtering dan Pagination

Dalam pembuatan sebuah *dashboard*, fitur untuk melakukan *filter* dan *pagination* selalu diterapkan. Salah satu manfaatnya adalah untuk memudahkan pengguna dalam membaca dan mencari data. Berdasarkan pembelajaran mengenai *filtering* dan *pagination* yang telah dilakukan, penerapan fitur tersebut menjadi lumrah dan menjadi satu kesatuan saat membuat master *dashboard* baru.

A Fitur Filtering Data Dashboard pada Tabel

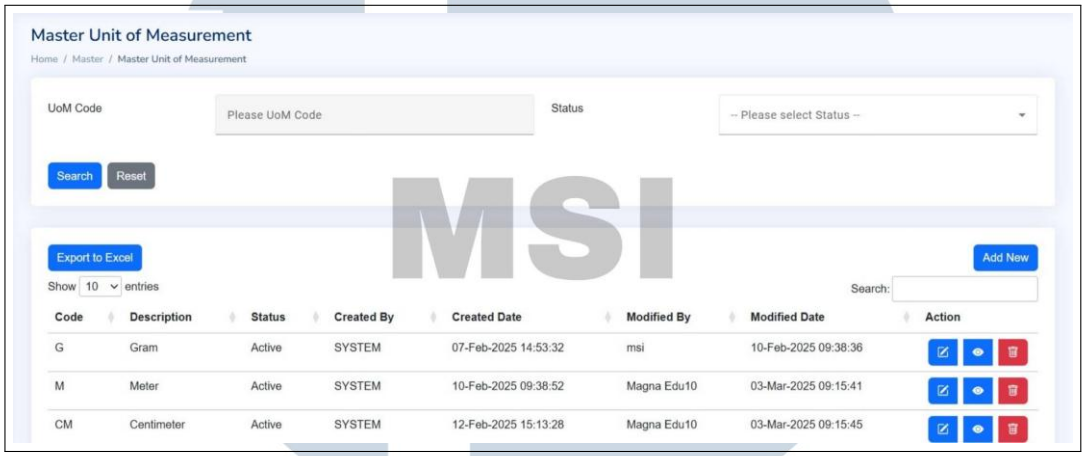
Proses filter dijalankan saat *user* ingin menampilkan atau mencari beberapa data saja dari banyaknya kumpulan data. Meskipun hanya melakukan *filtering* dan *reset*, terdapat proses pemanggilan ke sistem *back-end* juga. Berikut adalah *flowchart* untuk menjelaskan alur *filter* dan *reset*.



Gambar 3.43. Flowchart Search dan Reset Data pada Monitoring Table

Gambar 3.43 menjelaskan bahwa saat *user* menekan tombol **Search**, yang terjadi adalah kata kunci (kode atau deskripsi) yang diketik dikirimkan ke *back-*

end, jika input data yang dicari tersedia, maka akan menampilkan semua data yang mengandung kata kunci tersebut. Jika tidak ada, maka akan menampilkan ”*No Data Available on Table*”. Saat menekan tombol **Reset**, yang terjadi adalah mengirimkan permintaan ke *back-end* tanpa *filter* apa pun, sehingga akan mengembalikan semua data tanpa terkecuali. Berikut adalah proses *filter* secara langsung yang akan menggunakan Master UoM.

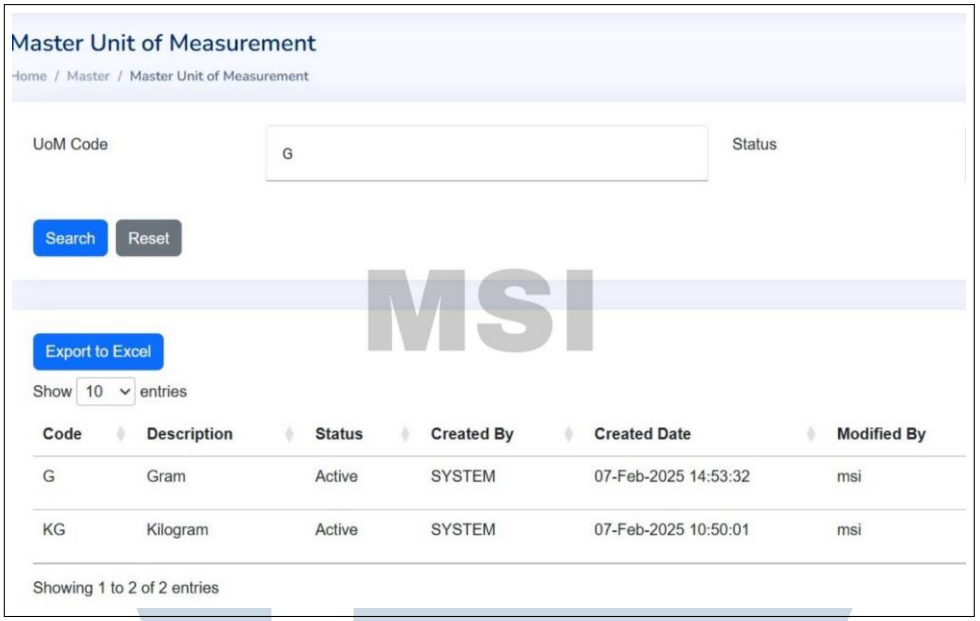


Gambar 3.44. *Show All UoM Data*

Gambar 3.44 menunjukkan tampilan yang tidak memiliki *filter* atau pencarian data tertentu. Data masih ditampilkan secara lengkap mulai dari *code, description, status, created by, created date, modified by, & modified date*. Pada UoM, terdapat dua indikator yang digunakan untuk mencari data, yaitu *code* dan *status*.

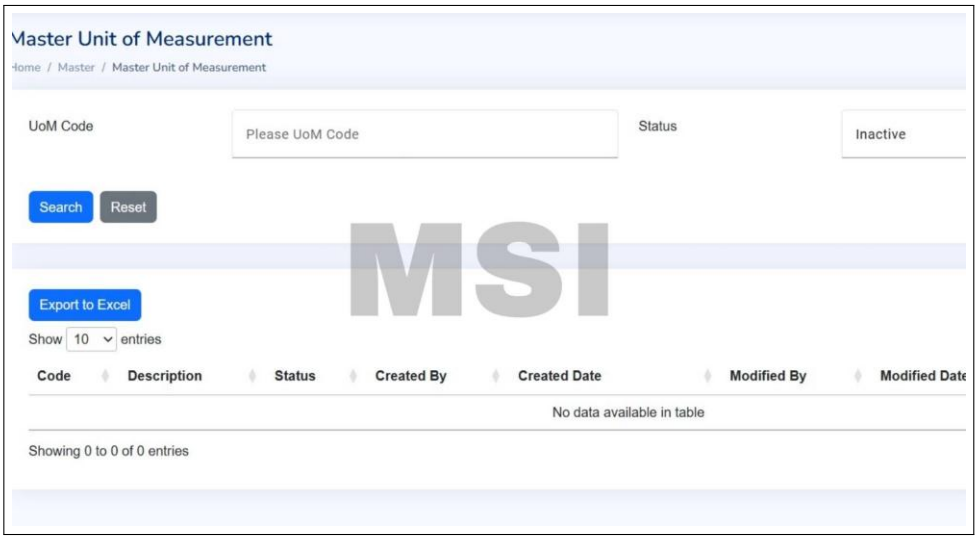
Alur proses penyaringan (*filter*) berdasarkan kata kunci (*keyword*) untuk kolom *code* dan *status*. Pada contoh master UoM, *code* dapat berisi nilai seperti ”m”, ”km”, ”cm”, ”l”, dan sebagainya, dengan *status* yang terdiri dari aktif dan nonaktif.

Saat *user* ingin melakukan pencarian, mereka cukup ketik kata kunci yang ingin dicari. Sistem kemudian akan melakukan pencocokan terhadap data di dalam *database* menggunakan *query* SQL yang telah disusun. Contoh dari implementasi pencarian dapat dilihat pada Gambar 3.45 dan Gambar 3.46.



Gambar 3.45. *Search UoM Code*

Pencarian *code* pada Gambar 3.45 menggunakan klausa ILIKE dengan wildcard (%ILIKE%) yang memungkinkan pencarian sebagian dengan *case-insensitive*. Dengan demikian, aplikasi akan mengembalikan semua data yang mengandung kata kunci tersebut ke dalam *datatable*.



Gambar 3.46. *Search UoM Status*

Namun, pencarian status pada Gambar 3.46 memiliki sedikit perbedaan. Hal tersebut karena *status* disimpan sebagai *Integer* atau angka dalam *database* yaitu berupa 0 dan 1. Status dengan angka 0 berarti nonaktif dan angka 1 berarti aktif.

Maka dari itu, pencarian pada SQL tidak menggunakan klausa ILIKE, melainkan langsung menggunakan simbol sama dengan (=). Pada Gambar 3.46, tidak ada data yang diatur berstatus nonaktif, sehingga tertulis keterangan *"No Data Available in Table"*.

Berikut adalah beberapa potongan kode mengenai cara kerja fitur *filtering* dalam suatu tabel.

```
this.formSearch = this.formBuilder.group({
  code: [null],
  uomFrom: [null],
  uomTo: [null],
  status: [null],
});
```

Gambar 3.47. *Set Value* Awal sebagai Null

Gambar 3.47 menunjukkan objek `formSearch` dibuat menggunakan `formBuilder.group()` pada saat komponen dijalankan. Objek-objek tersebut berisi `code, uomFrom, uomTo`, dan `status`. Masing-masing *field* diisi dengan nilai awal **null**, yang akan diubah berdasarkan input dari pengguna.

```
getParamFilter() {
  interface search {
    code?: string;
    uomFromDesc?: string;
    uomToDesc?: string;
    status?: number;
  }

  const search: search = {};

  if (this.s.code.value != null && this.s.code.value != '') {
    search.code = this.s.code.value;
  }

  if (this.s.uomFrom.value != null && this.s.uomFrom.value != '') {
    search.uomFromDesc = this.s.uomFrom.value;
  }

  if (this.s.uomTo.value != null && this.s.uomTo.value != '') {
    search.uomToDesc = this.s.uomTo.value;
  }

  if (this.s.status.value != null) {
    search.status = this.s.status.value;
  }

  return search;
}
```

Gambar 3.48. Fungsi untuk Membaca Input

Pengambilan nilai filter terjadi pada Gambar 3.48. Ketika pengguna menekan tombol ”*Search*” yang terdapat pada Gambar 3.49, fungsi `getParamFilter()` akan dipanggil. Fungsi ini membaca setiap nilai dari form control, lalu menyusunnya ke dalam objek *search*. Hanya *field* yang tidak memiliki nilai (tidak kosong) yang akan disertakan dalam objek filter. Misalnya, jika pengguna hanya mengisi `code` dan `status`, maka hanya dua filter tersebut yang akan dikirim. Ini mencegah pengiriman data kosong agar tidak terjadi `NullPointerException`.

```
onClickSearch() {
  this.isSearch = true;
  this.isAdd = false;
  this.isEdit = false;
  this.isView = false;
  this.isDelete = false;

  let urlAPI = apienv.moduleRest001 + apienv.uomConversionRest;
  let filter = this.getParamFilter();
  let param = this.getParam(null, null, null, null, filter);

  this.callAPI(urlAPI, param);
}
```

Gambar 3.49. Fungsi untuk Memanggil API

Setelah objek disusun, fungsi `getParam()` pada Gambar 3.49 akan membungkus *filter* ke dalam format parameter API. Nilai lain seperti `offset, limit, order`, dan `sort` bisa diisi null jika tidak diperlukan. Parameter akan dikirim ke *back end* menggunakan fungsi `callAPI(urlAPI, param)`. Di tahap ini data *filter* akan diproses di *back-end* lebih lanjut.

```
if (searchFilter.containsKey(key:"code")) {
    sql += " AND a.vcode ILIKE '%" + searchFilter.get(key:"code") + "%' ";
}
if (searchFilter.containsKey(key:"uomFromDesc")) {
    sql += " AND tu1.vdesc ILIKE '%" + searchFilter.get(key:"uomFromDesc") + "%' ";
}
if (searchFilter.containsKey(key:"uomToDesc")) {
    sql += " AND tu2.vdesc ILIKE '%" + searchFilter.get(key:"uomToDesc") + "%' ";
}
if (searchFilter.containsKey(key:"status")) {
    sql += " AND a.nstatus = " + searchFilter.get(key:"status") + " ";
}
```

Gambar 3.50. Pengujian Kata Kunci pada *Back-End*

Di sisi *back end* (Java DAO), parameter *filter* diterima sebagai `Map<String, Object>` dan diproses menggunakan method `containsKey` untuk memeriksa *field* yang digunakan. Jika ada, maka `SQL WHERE` akan ditambahkan dengan kondisi yang sesuai. Misalnya *value input* `vcode ILIKE` "%*value*%".

B Fitur Pagination Dashboard pada Tabel

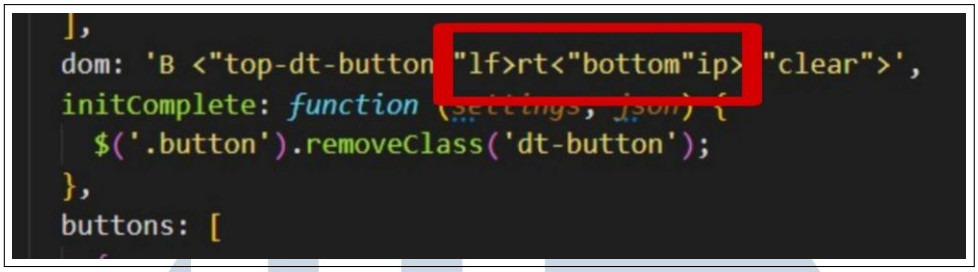
Bagian kedua adalah membahas mengenai *Pagination* yang digunakan oleh seluruh halaman master dalam proyek DMS. Berbeda dengan fitur *List of Values* yang menggunakan *server-side pagination* melalui parameter offset dan limit, komponen master seperti contohnya UoM Conversion menggunakan *client-side pagination* yang dikonfigurasi langsung melalui properti `this.dtOption` (tidak harus diberi nama dtOption).

```
this.dtOption = {
  pageLength: 10,
  processing: true,
  autoWidth: true,
  order: [],
  responsive: {
    details: {
```

Gambar 3.51. Mengatur Jumlah Data per Halaman

Gambar 3.51 menjadi salah satu kunci pada konfigurasi tersebut karena terdapat properti `pageLength` yang mengatur jumlah data per halaman (misalnya 10 baris). Jika `pageLength` diatur dengan jumlah tertentu, maka sisa data akan dibagi ke halaman berikutnya. Pengguna juga tetap dapat mengubah jumlah ini secara manual melalui *dropdown* yang disediakan oleh *DataTables*.

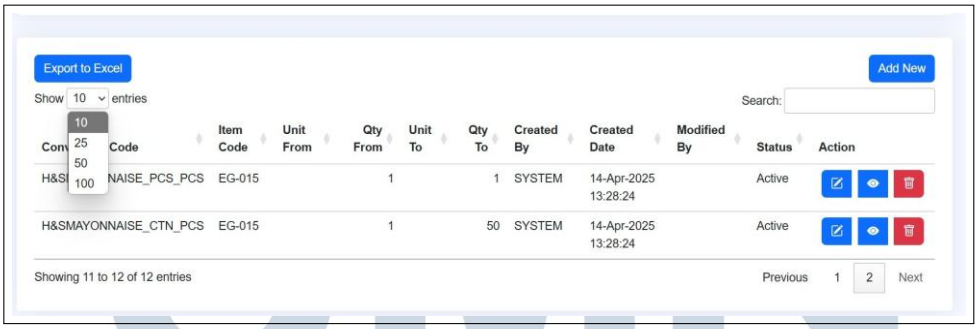
Properti DOM adalah sebuah konfigurasi untuk mengatur struktur dan posisi dari elemen-elemen kontrol tabel (seperti kotak pencarian, info jumlah data, dan tombol *pagination* di dalam halaman HTML. Sintaksis dari DOM adalah fitur spesifik dari *library* JavaScript bernama *jQuery DataTables*. Fitur *pagination* pada master dalam proyek DMS dibuat dengan konfigurasi tersebut.



Gambar 3.52. Mengatur Jumlah Data per Halaman

Properti DOM yang digunakan pada Gambar 3.52 penting untuk menentukan struktur dan posisi elemen-elemen *DataTables* di dalam halaman. Beberapa elemen penting yang diatur oleh konfigurasi DOM antara lain:

- l: menampilkan *dropdown* untuk memilih jumlah data per halaman.
- f: menampilkan *search box* untuk melakukan pencarian data.
- i: menampilkan informasi jumlah data yang sedang ditampilkan.
- p: menampilkan kontrol *pagination*, seperti tombol *next/previous*.



Gambar 3.53. Contoh Tampilan DataTable dengan *Pagination*

Pada Gambar 3.53, *dropdown* pada ujung kiri atas menunjukkan bagian yang diatur oleh `pageLength` dan properti dom ”l”, sehingga pengguna dapat memilih banyak data yang ingin ditampilkan dalam satu halaman tersebut. Lalu pada ujung kiri bawah yang bertuliskan ”*Show 11 to 12 of 12 entries*” diatur dalam properti dom ”i”, berfungsi untuk menampilkan informasi jumlah data. Pada bagian ujung kanan atas, terdapat *search box* yang diatur oleh properti dom ”f”, dan yang terakhir adalah properti ”p” yang menampilkan kontrol seperti tombol *next/previous*.

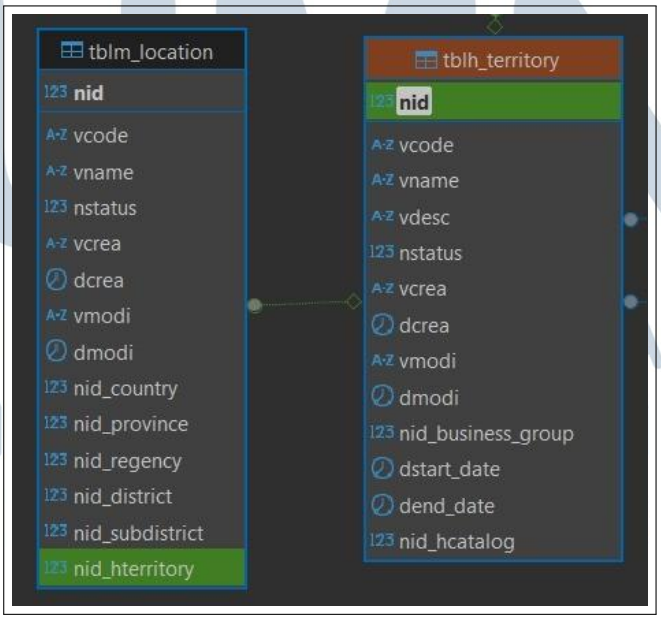
3.3.7 Menerapkan Fitur Global Checkbox pada Master Territory

Tugas selanjutnya adalah untuk menerapkan fitur global *checkbox* pada Master Territory. Tidak hanya menerapkan *front-end* dan *back-end*, juga diperlukan pengetahuan mengenai relasi antara Master Territory dan Master Location untuk memenuhi kriteria logika bisnis yang diperlukan.

A Relasi antara Master Territory dan Master Location

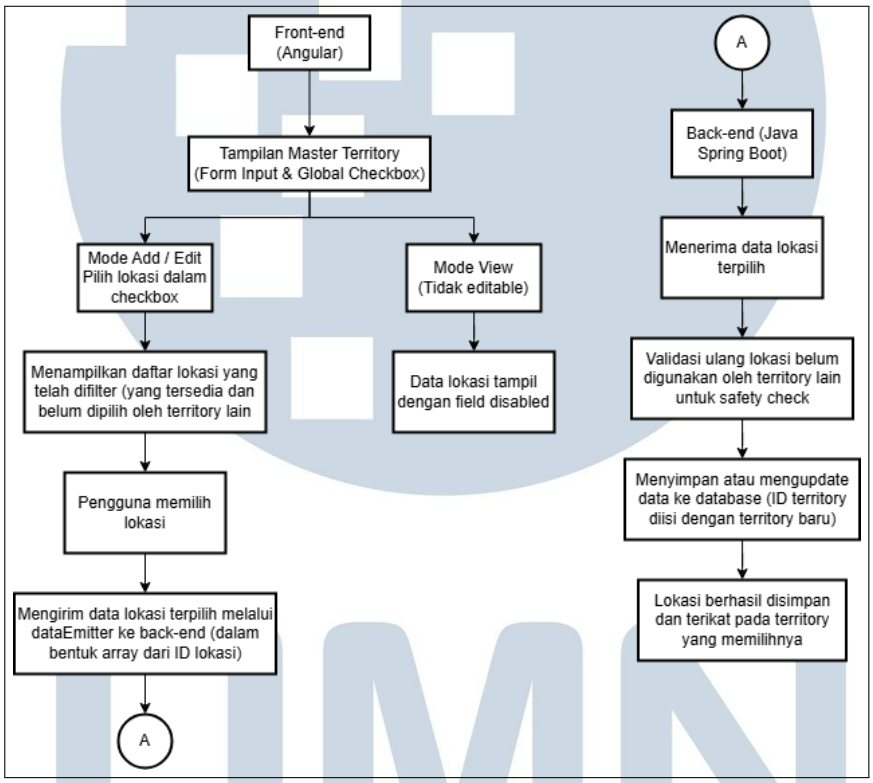
Dalam *distribution management system*, *territory* dan *location* memiliki hubungan untuk mendefinisikan area operasi distributor atau agen tertentu. Setiap wilayah (*territory*) terdiri dari satu atau lebih lokasi (*location*). Namun, satu lokasi hanya boleh dimiliki oleh satu wilayah untuk mencegah konflik alokasi area penjualan. Sehingga dengan hubungan tersebut, jika suatu *location* telah terhubung dengan suatu *territory*, maka *location* tersebut tidak dapat dipilih oleh *territory* lain.

Aturan ini diterapkan untuk menjaga konsistensi data dan menghindari tumpang tindih wilayah operasional yang dapat menyebabkan kesalahpahaman dalam proses distribusi. Dengan demikian, setiap *territory* memiliki kontrol penuh terhadap *location* yang menjadi cakupannya, sehingga distribusi produk dapat berjalan lebih efektif dan terarah. Relasi dari kedua tabel tersebut dapat dilihat pada Gambar 3.54.



Gambar 3.54. Relasi antara Tabel *Location* dan *Territory*

Gambar 3.54 menunjukkan relasi dari kedua tabel. Fokus utama dalam tabel-tabel tersebut adalah menghubungkan `nid hterritory` sebagai *header* pada *location*. Sehingga saat `nid hterritory` tidak bernilai **NULL**, maka lokasi tersebut tidak dapat digunakan lagi di wilayah lain. Kolom-kolom lain yang terdapat di tabel meliputi `vcode, vname, vdesc, nstatus`, dan lain sebagainya sebagai standar. Berikut adalah bagan desain yang akan ditampilkan untuk menggambarkan proses yang terjadi pada Master Territory secara keseluruhan.



Gambar 3.55. Bagan Desain untuk Master Territory

Gambar 3.55 adalah bagan desain untuk menjelaskan menu Master Territory secara keseluruhan. Sejak awal, daftar lokasi yang ditampilkan saat ingin menamabah dan memodifikasi data adalah data yang telah dilakukan *filter* terlebih dahulu agar hanya lokasi dengan `nid hterritory` bernilai NULL. Oleh karena itu, pengguna tidak mungkin untuk memilih lokasi yang sudah digunakan sebelumnya oleh *territory* lain. Data yang dipilih dikirim ke *back-end* dalam bentuk *array*, kemudian logika *back-end* memperbarui lokasi tersebut di *database* pada kolom `nidhterritory` dengan ID yang baru.

B Integrasi Global Checkbox pada Master Territory

Fitur *global checkbox* telah dibuat oleh seorang senior developer, sehingga hanya perlu dilakukan integrasi secara mandiri terhadap suatu master jika dibutuhkan. Integrasi fitur tersebut dapat dilihat pada potongan kode berikut.

```
<app-input-cb-table-server-side-global labelClass="none" fieldClass="w-100" labelString="Location"
  name="Territory Location" (dataEmitter)="onSelectionLocation($event)" [mandatory]="true"
  [column]="['id', 'name']" [value]="getFormDetail('detailLocationId').value"
  [filterValue]="a.inputId.value" [disable]="isView">
</app-input-cb-table-server-side-global>
```

Gambar 3.56. Kode HTML untuk *Checkbox*

Kode HTML pada Gambar 3.56 memiliki **Location** sebagai label dan menyimpan data sebagai **Territory Location** yang akan diarahkan ke *back-end*. Fungsi `dataEmitter` adalah untuk mengirimkan data yang dipilih, lalu `value` untuk menyimmpan data yang dipilih. Terdapat juga `filterValue` yang menyaring data berdasarkan input dan `disable` atau tidak dapat diubah saat tampilan dalam mode **view**.

```
onSelectionLocation(event: any) {
  Let location;
  if (event.length > 0) {
    location = event.map((x) => x.id);
  } else {
    location = null;
  }

  this.getFormDetail('detailLocationId').setValue(location);
}
```

Gambar 3.57. Fungsi `onSelectionLocation` pada `dataEmitter`

Pada Gambar 3.57, saat pengguna memilih lokasi pada checkbox, data tersebut dikirim melalui `dataEmitter`. Fungsi `onSelectionLocation(event)` adalah untuk menangkap *event* tersebut, sehingga jika data dipilih, maka *location* akan berupa array dari id, jika tidak ada, maka *location* akan diset **NULL**. Data yang telah dipilih disimpan dalam `detailLocationId`.

Entitas pada *back-end* yang bernama `voMasterTerritory` merepresentasikan data utama dari suatu *territory*. Selain menyimpan data kode dan nama *territory*, tersimpan juga data lokasi terkait

melalui atribut `territory_detail`, yang bertipe list dan berisi objek `voMasterTerritoryDetail`.

Sementara itu, `voMasterTerritoryDetail` digunakan untuk menyimpan detail lokasi seperti **locationId, locationName**, dan **locationCode**. Pendekatan ini langsung terhubung ke entitas utama tadi yaitu *territory* dalam bentuk list. Setelah *territory* utama dikumpulkan, daftar lokasi yang terkait pada *territory* tersebut juga akan diambil menggunakan metode `getLocationsByTerritory(territoryId)`. Fungsi ini hanya akan mengamil lokasi yang telah dimiliki *territory* tersebut, yaitu data lokasi yang memiliki nilai `nid_hterritory` yang sesuai.

Gambar 3.58. Struktur Data Master Territory

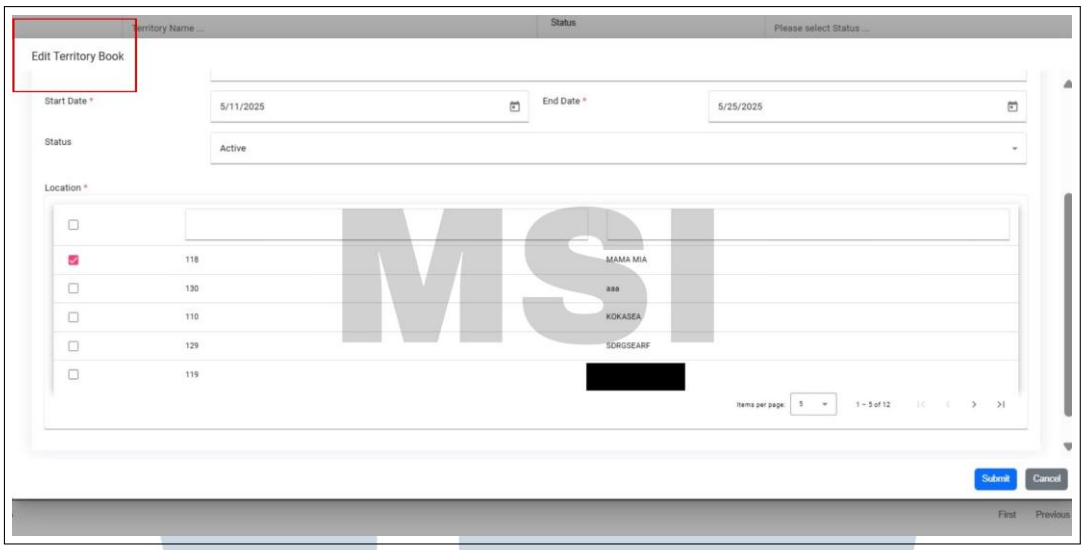
Gambar 3.58 merupakan implementasi konkret dari struktur data `voMasterTerritory` dan `voMasterTerritoryDetail`. Struktur JSON tersebut memperlihatkan bagaimana `voMasterTerritory` menyimpan atribut utama seperti `code`, `name`, `desc` dan `business_group_id`, serta `territory_detail` dalam bentuk list `voMasterTerritoryDetail` yang berisi data lokasi (`locationId`, `locationName`, `locationCode`). Dengan struktur tersebut, data lokasi dapat diatur dengan lebih mudah karena terhubung langsung engan entitas *territory*.

C Validasi untuk Memenuhi Kriteria Logika Master Territory

Validasi *back-end* digunakan untuk memastikan hanya lokasi yang belum digunakan *territory* lain yang dapat muncul dan dipilih. Validasi ini dilakukan untuk mengantisipasi konflik ketika ada beberapa *user* mengakses data yang sama dalam waktu bersamaan. Selain itu, proses penyimpanan data lokasi untuk pertama kali (saat nid_hterritory masih NULL), sistem akan langsung melakukan **UPDATE** pada *query* agar lokasi terikat ke *territory* yang baru. Jika tidak, maka lokasi tersebut akan tetap terlihat tersedia dan memiliki risiko dipilih oleh *territory* lain. Validasi ini juga berlaku saat ingin melakukan modifikasi data.

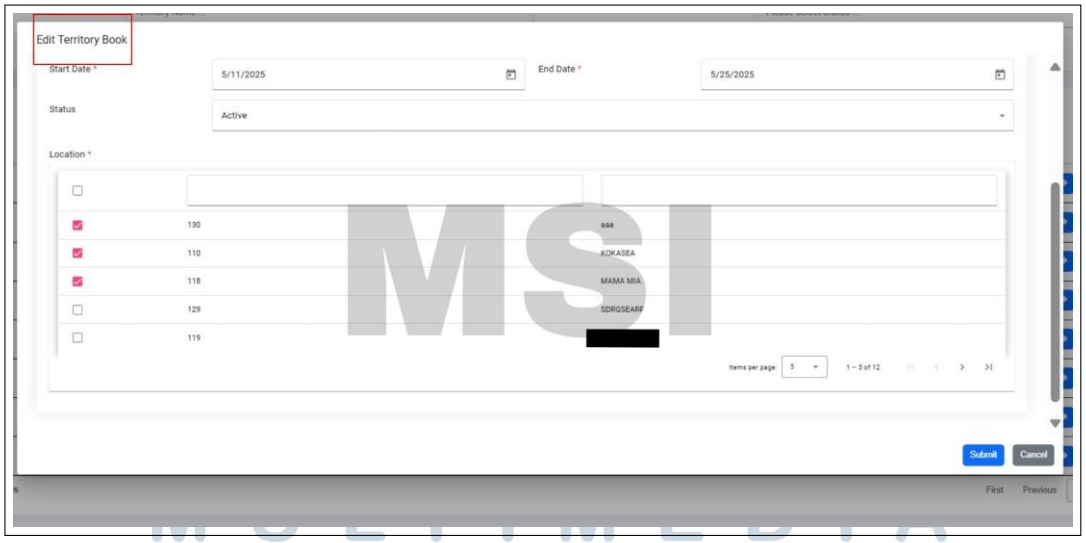
Gambar 3.59. Contoh *Add* Data pada Checkbox

Gambar 3.59 memberikan contoh saat ingin menambahkan data Master Territory. Pada tahap pertama, pengguna akan memasukkan data-data yang diperlukan serta memilih lokasi dalam checkbox tersebut, lokasi dapat dipilih lebih dari satu. Jika data lokasi telah dipilih dan pengguna ingin menambahkan data baru lagi, maka lokasi terpilih tadi tidak akan tampil dalam tabel checkbox tersebut. Pada contoh ini, akan digunakan lokasi dengan **ID: 118** dan **Location Name: MAMA MIA**.



Gambar 3.60. Contoh *Edit* Data pada Checkbox

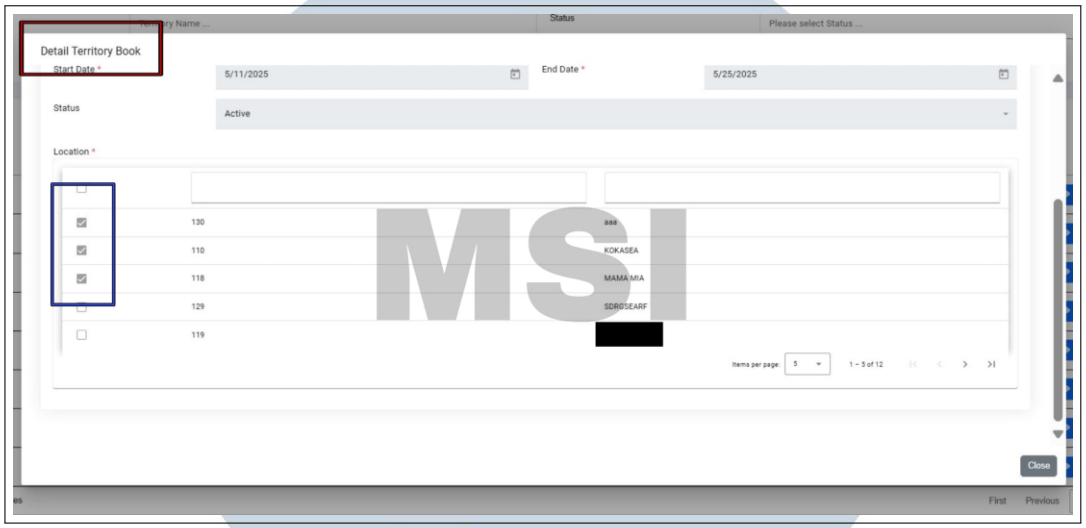
Gambar 3.60 menunjukkan tampilan saat ingin melakukan *edit* pada data yang telah ditambahkan tadi, yaitu dengan **ID: 118** dan **Location Name: MAMA MIA**. Tampilan *edit* menunjukkan bahwa lokasi terpilih akan langsung otomatis tercantum (*checked*) dalam tabel. Sehingga jika ingin menambahkan daftar lokasi, maka pengguna hanya perlu melakukan *checklist* pada data yang diinginkan.



Gambar 3.61. Contoh *Edit* Data pada Checkbox

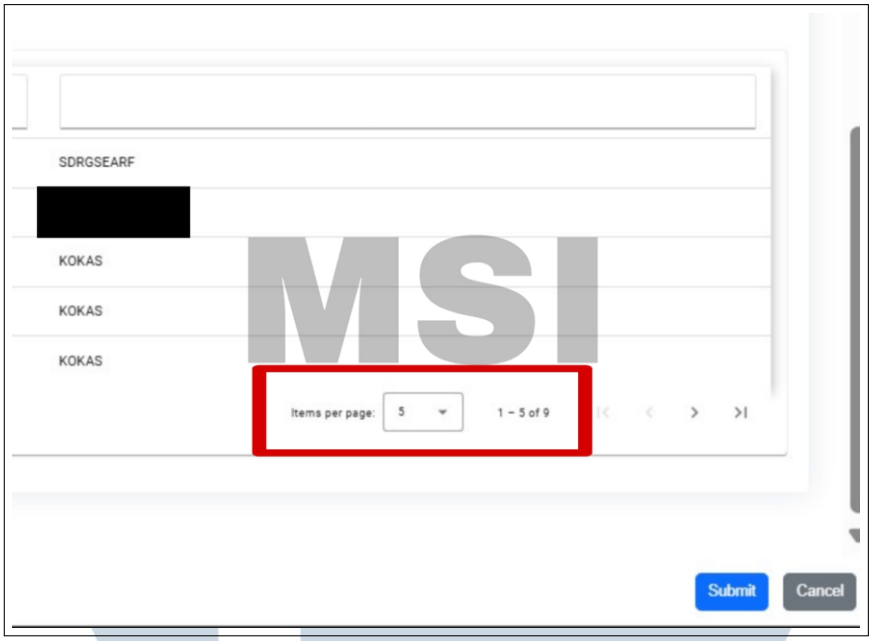
Pada Gambar 3.61, setelah melakukan *checklist* pada data-data lokasi yang diinginkan dan membuka kembali dialog *edit*, maka sama seperti sebelumnya, data-data terpilih akan otomatis tercantum *checked* pada tabel. Serupa seperti fitur *add*,

jika pengguna membuka dialog *edit* untuk data lain, maka data lokasi terpilih yang berada di Gambar 3.61 tidak akan tampil kembali di dialog lain, contohnya dapat dilihat pada Gambar 3.63.



Gambar 3.62. Contoh *View* Data pada Checkbox

Gambar 3.62 adalah tampilan untuk melihat atau *view* data. Dalam contoh ini, data yang ingin dilihat adalah data sebelumnya yang awalnya hanya memiliki **ID: 118** dan **Location Name: MAMA MIA**. Setelah sebelumnya dimodifikasi, maka data lokasi yang terpilih menjadi berjumlah tiga, yaitu **([ID: 118, Location Name: MAMA MIA], [ID: 119, Location Name: KOKASEA], [ID: 130, Location Name: aaa])**. Tampilan juga terlihat *disabled* agar tidak dapat diubah, hal tersebut karena fungsi `disabled` pada kode HTML sebelumnya.



Gambar 3.63. Perhitungan Jumlah Data pada Tabel Checkbox

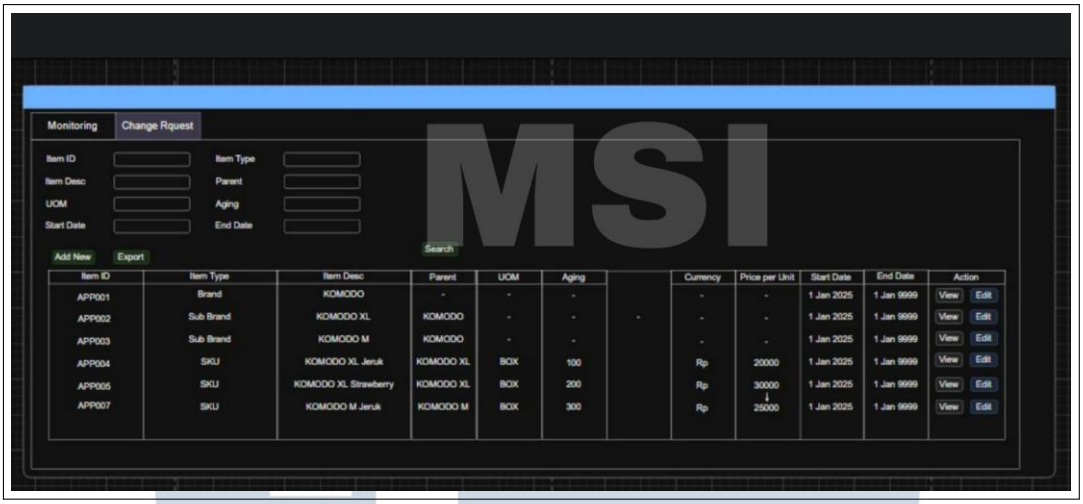
Gambar 3.63 adalah tampilan saat ingin menambahkan data baru setelah tadi pengguna telah menambahkan tiga data lokasi pada suatu *territory*. Sebelumnya, total data yang tercatat berjumlah 12 data, namun setelah tiga lokasi telah disimpan dalam suatu *territory*, maka data berkurang menjadi 9. Tidak hanya saat ingin menambahkan data baru, ketika pengguna ingin melakukan *edit* dan membuka dialog *edit* pada data *territory* lain, maka data-data lokasi yang tampil hanyalah lokasi yang tersedia dan belum terpilih di *territory* lain.

3.3.8 Memperbaiki Tampilan Master Item

Dalam *dashboard Distribution Management System*, terdapat master item yang berfungsi untuk menyimpan semua data barang, baik yang akan dijual dalam katalog maupun tidak.

A Tampilan Awal Master Item

Pada awalnya, master item memiliki tampilan dengan format *monitoring* seperti yang terlihat pada Gambar 3.64.

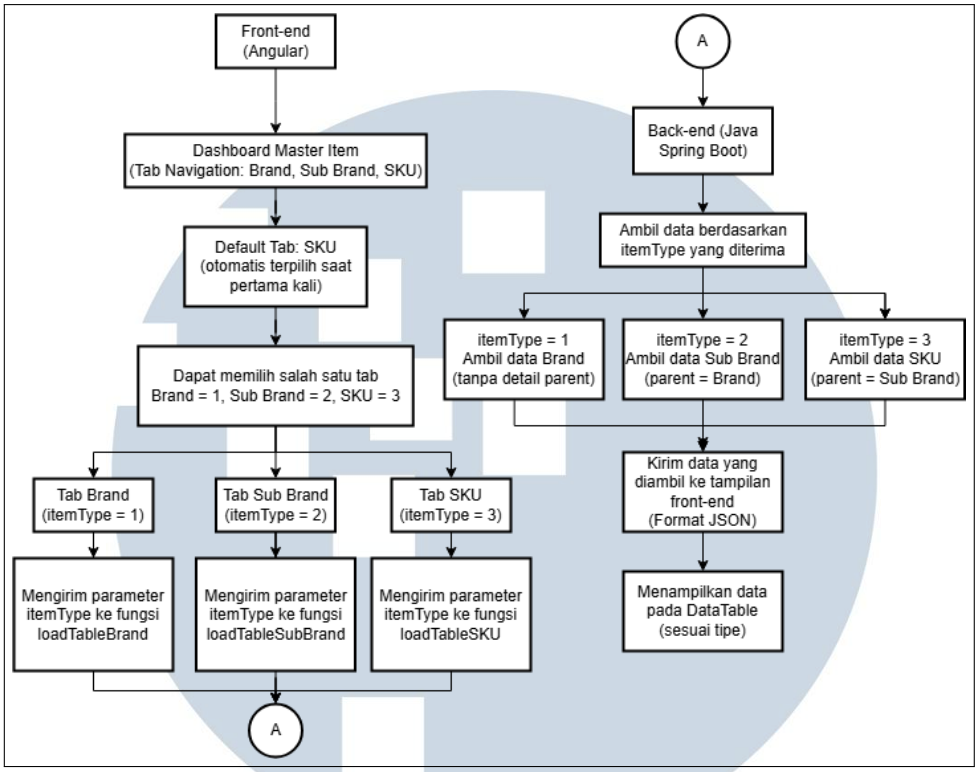


Gambar 3.64. Tampilan Master Item Sebelum Perbaikan

Pada Gambar 3.64 terlihat bahwa *filter search* pada master item akan mencari data berdasarkan **Item ID, Item Type, Item Desc, Parent, UOM, Aging, Start Date**, dan **End Date**. Namun, tabel *monitoring* tersebut masih menampilkan seluruh data barang dalam satu tabel, baik yang bertipe Brand, Sub-brand, dan SKU. Akibatnya, tampilan menjadi tidak terstruktur dan membingungkan bagi pengguna.

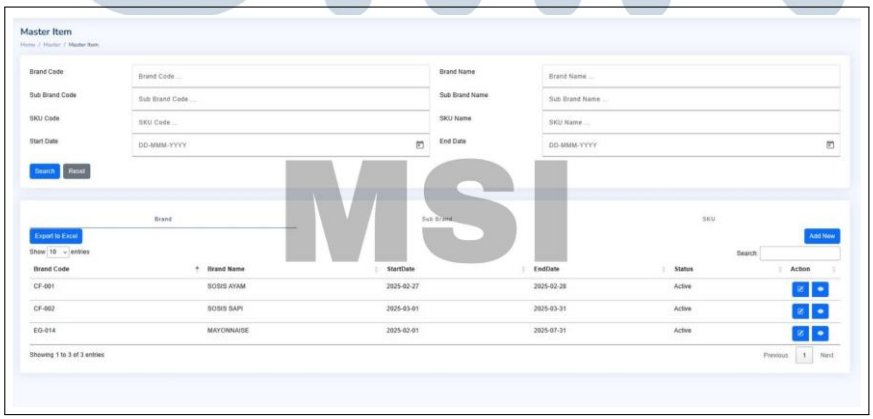
B Tampilan Master Item Setelah Perbaikan

Permasalahan tersebut diatasi dengan melakukan perbaikan untuk memisahkan setiap tipe barang ke dalam tab tersendiri, sehingga pengguna dapat lebih mudah mengelola data Brand, Sub-brand, dan SKU. Berikut adalah gambaran umum mengenai perbaikan yang dilakukan dengan menggunakan bagan desain.



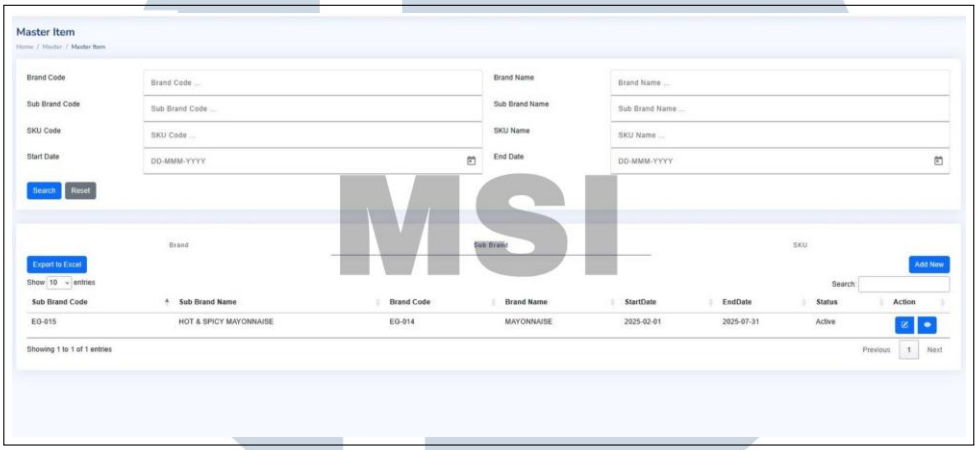
Gambar 3.65. Bagan Desain pada Menu Master Item

Gambar 3.65 menjelaskan *dashboard* Master Item yang memiliki tab navigasi untuk memiliki kategori data item (Brand, Sub Brand, SKU). Saat pengguna memilih salah satu tab, parameter `itemType` dikirim ke fungsi **loadTable** untuk memanggil data yang sesuai. *Back-end* menerima *request* tesebut dan menjalankan *query* SQL untuk mengambil data sesuai kategori. Data yang sudah didapatkan dikembalikan ke *front-end* dalam bentuk JSON dan data siap ditampilkan sesuai tab yang dipilih.



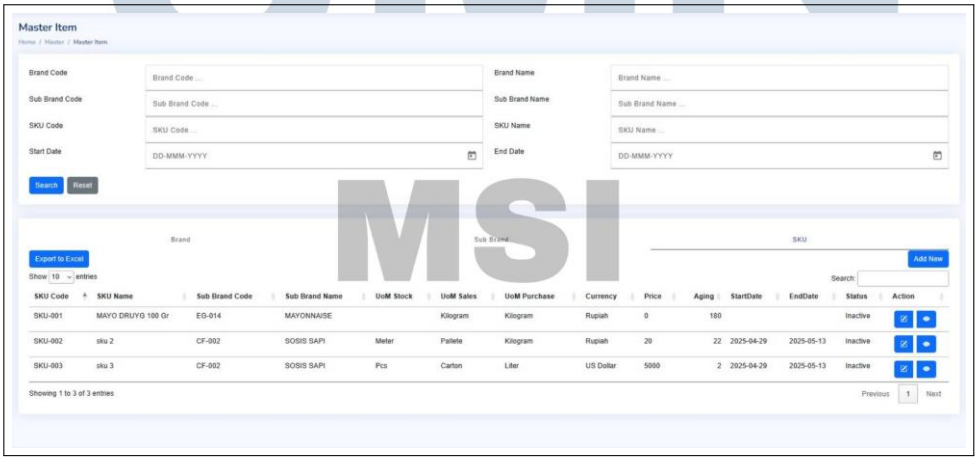
Gambar 3.66. Master Item Brand

Gambar 3.66 menaampilkan data yang difokuskan hanya untuk tipe **Brand**. Pemisahan tampilan ini bertujuan untuk memudahkan pengguna dalam mencari dan mengelola data brand secara spesifik, tanpa tercampur dengan data sub-brand atau SKU. Fitur CRUD juga diimplementasikan pada tab ini, sehingga saat menambahkan data, sistem akan memastikan bahwa data tersebut akan disimpan sebagai brand.



Gambar 3.67. Master Item Sub-brand

Gambar 3.67 menampilkan data yang berfokus pada tipe **Sub-brand**. Kolom-kolom yang ditampilkan meliputi **Sub Brand Code, Sub Brand Name, Brand Code, Brand Name, Start Date, End Date**, dan **Status**. Pemisahan tampilan sub-brand ini memungkinkan pengguna untuk mengelola secara lebih terstruktur dan detail. Perlu diperhatikan bahwa data sub-brand merupakan *child* dari brand, sehingga setiap sub-brand harus memiliki *brand parent* yang sesuai.



Gambar 3.68. Master Item SKU

Gambar 3.68 menunjukkan data **SKU** yang ditampilkan secara terpisah dari brand dan sub-brand. Kolom yang ditampilkan meliputi **SKU Code, SKU Name, Sub Brand Code, Sub Brand Name, UoM Stock, UoM Sales, UoM Purchase, Currency, Price, Aging, Start Date, End Date**, dan **Status**.

Pemisahan data SKU ini sangat penting karena SKU biasanya memiliki detail informasi yang lebih kompleks dibandingkan dengan brand atau sub-brand. Tampilan SKU difokuskan untuk menampilkan informasi hingga level terkecilnya,, yaitu SKU, agar pengguna dapat lebih mudah melakukan pengelolaan data terkait stok barang, harga, dan masa aktif SKU tersebut.

C Alur Pemisahan Kategori pada Item dari Front-end hingga Back-end

Pemisahan tampilan item tidak dilakukan dengan membuat tiga tabel terpisah untuk brand, sub-brand, dan SKU. Sebaliknya, implementasi dilakukan melalui pengiriman parameter tipe item (`itemType`) dari *front-end* ke *back-end*. Proses pada *back-end* akan melihat tipe item apa yang dikirim, jika sesuai maka akan dikembalikan tampilannya sesuai dari parameter yang dikirim. Detail kode implementasi pemisahan dapat dilihat pada gambar-gambar berikut.

```
<mat-tab-group [(selectedIndex)]="selectedTabIndex" (selectedIndexChange)="onTabChange($event)">
    <mat-tab label="Brand">
        <ng-container *ngIf="tabLoaded.brand">
            <div class="datatable-wrapper">
                <table datatable [dtOptions]="dtOptionBrand" id="dtBrand" class="table dtBrand"
                    width="100%">
                </table>
            </div>
        </ng-container>
    </mat-tab>
    <mat-tab label="Sub Brand">
        <ng-container *ngIf="tabLoaded.subbrand">
            <div class="datatable-wrapper">
                <table datatable [dtOptions]="dtOptionSubBrand" id="dtSubBrand" class="table dtSubBrand"
                    width="100%">
                </table>
            </div>
        </ng-container>
    </mat-tab>
    <mat-tab label="SKU">
        <ng-container *ngIf="tabLoaded.sku">
            <div class="datatable-wrapper">
                <table datatable [dtOptions]="dtOptionSKU" id="dtSKU" class="table dtSKU" width="100%">
                </table>
            </div>
        </ng-container>
    </mat-tab>
</mat-tab-group>
```

Gambar 3.69. Pemisahan Tampilan Item pada HTML

Pada Gambar 3.69, tampilan HTML menunjukkan terdapat tiga tab utama yaitu **Brand, Sub Brand, SKU**. Masing-masing tab berisi elemen `<table>` yang akan digunakan untuk menampilkan data sesuai tipe item. Saat tab dipilih,

atribut `[dtOptions]` akan berfungsi untuk mengikat *DataTables* dengan opsi yang sesuai (brand, sub-brand, atau SKU). Pemisahan tab tersebut menggunakan `mat-tab-group` dari Angular Material. Fungsi `selectedIndexChange` akan memanggil `onTabChange($event)` yang berfungsi menangani logika pergantian tab.

```
ngOnInit(): void {
  this.tabLoaded.sku = true;
  let urlAPI = apienv.masterItemRest;
  let filter = this.getFilter();
  let paramSKU = this.getParam(null, null, null, null, { ...filter, itemType: 3 })
  this.loadTableSKU(urlAPI, paramSKU, 'sku');
  this.setPreparationLOV();
}
```

Gambar 3.70. Pengaturan Tab SKU sebagai *Default*

Pada Gambar 3.70, metode `ngOnInit()` akan memuat tab SKU pertama kali saat halaman pertama kali dibuka. Hal ini dilakukan melalui inisialisasi `this.tabLoaded.sku = true`. Parameter `paramSKU` dibentuk menggunakan metode `getParam()` yang menggabungkan filter pencarian dan tipe item `itemType: 3` untuk SKU. Setelah parameter siap, fungsi `loadTableSKU()` akan dipanggil untuk mengambil data SKU dari API dan menampilkannya pada tabel.

```
onTabChange(index: number) {
  this.selectedTabIndex = index;
  let urlAPI = apienv.masterItemRest;
  let filter = this.getFilter();

  if (index === 0 && !this.tabLoaded.brand) {
    this.tabLoaded.brand = true;
    const param = this.getParam(null, null, null, null, { ...filter, itemType: 1 });
    this.loadTableBrand(urlAPI, param, 'brand');
  } else if (index === 1 && !this.tabLoaded.subbrand) {
    this.tabLoaded.subbrand = true;
    const param = this.getParam(null, null, null, null, { ...filter, itemType: 2 });
    this.loadTableSubBrand(urlAPI, param, 'subbrand');
  } else if (index === 2 && !this.tabLoaded.sku) {
    this.tabLoaded.sku = true;
    const param = this.getParam(null, null, null, null, { ...filter, itemType: 3 });
    this.loadTableSKU(urlAPI, param, 'sku');
  }

  setTimeout(() => {
    const dtId = ['#dtBrand', '#dtSubBrand', '#dtSKU'][index];
    if ($.fn.DataTable.isDataTable(dtId)) {
      const dtInstance = $(dtId).DataTable();
      dtInstance.columns.adjust();
      dtInstance.page.len(dtInstance.page.len()).draw(false);
    }
  }, 300);
}
```

Gambar 3.71. Pengaturan Mengganti Tab

Gambar 3.71 menjelaskan logika pada metode `onTabChange()` yang menangani perubahan tab. Jika tab pertama (index 0) adalah **Brand**, maka parameter `itemType: 1` akan dikirim. Jika tab kedua (index 1) adalah **Sub Brand**, maka parameter `itemType: 2` akan dikirim. Jika tab ketiga (index 2) adalah **SKU**, maka parameter `itemType: 3` akan dikirim.

Pada setiap tab, fungsi `this.loadTableBrand()`, `this.loadTableSubBrand()`, atau `this.loadTableSKU()` akan dipanggil untuk mengambil data sesuai tipe item. Fungsi `setTimeout()` di bagian bawah kode memastikan bahwa *DataTables* akan diperbarui setiap kali tab berubah, sehingga tampilan data tetap konsisten dan tidak tumpang tindih antar tab.

Pemisahan tampilan menggunakan *tab-navigation* ini tidak hanya membuat tampilan lebih terstruktur tetapi juga meningkatkan efisiensi sistem karena hanya data pada tab yang aktif saja yang akan dimuat, sehingga beban pada *client-side* menjadi lebih ringan. Semua data dalam item dipanggil dengan API masterItemRest, yang berisi *query* seperti Gambar 3.72.

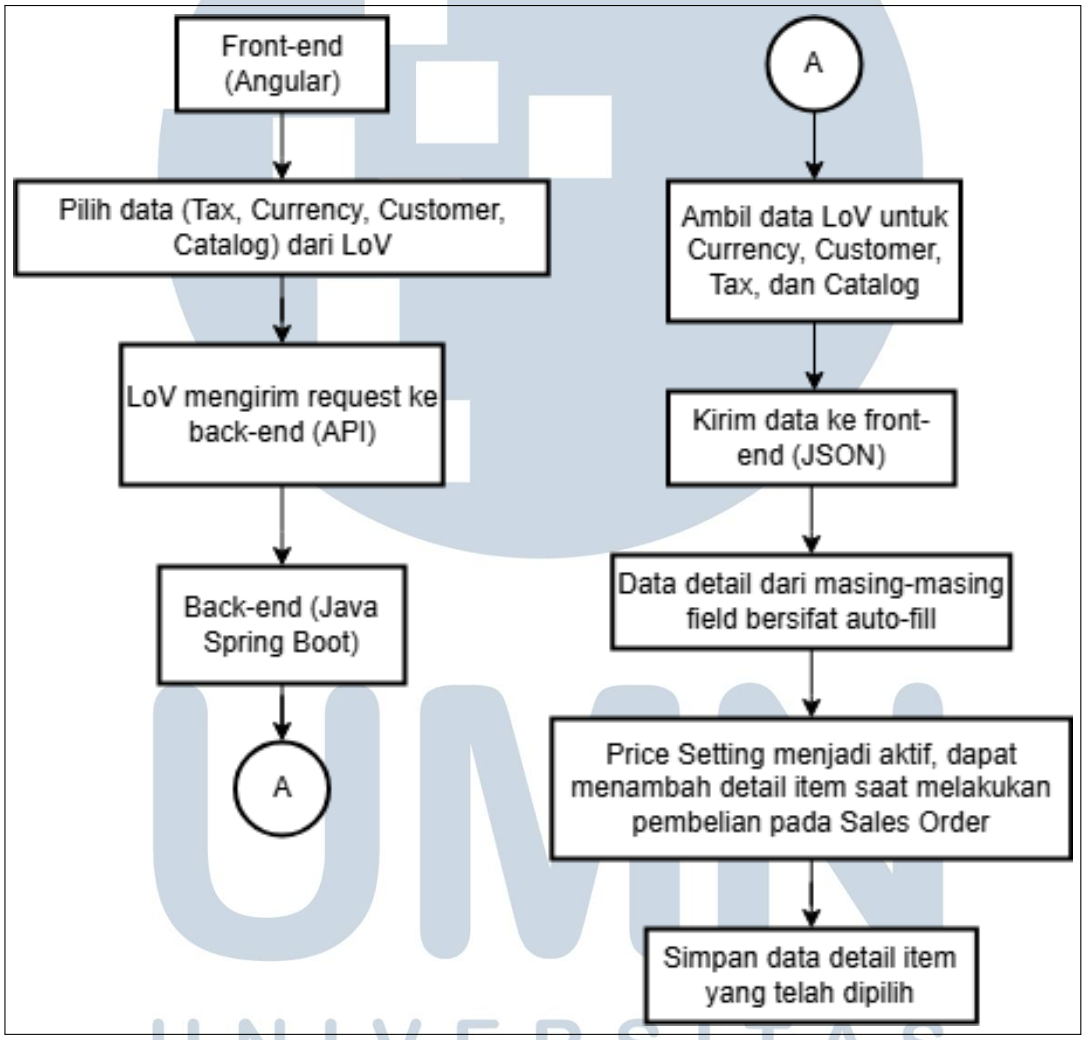
```
" SELECT "
 + " MSITEM.nid itemID, MSITEM.vitem_code itemCode, MSITEM.vitem_desc itemDesc "
 + " , MSITEM.vitem_type itemTypeCode, MSSETT.vdesc itemTypeDesc "
 + " , MSITEM.dstart_date, MSITEM.dend_date "
 + " , MSITEM.nparent_id parentID, PRITEM.vitem_code parentCode, PRITEM.vitem_desc parentDesc, PRITEM.vitem_type parentTypeId "
 + " , MSITEM.nid_uom_stock "
 + " , MSITEM.nid_currency, MSCURR.vname currencyDesc "
 + " , MSITEM.naging, MSITEM.nprice, MSITEM.nweight, MSITEM.nvolume "
 + " , MSITEM.nstatus, MSITEM.vcrea, MSITEM.dcrea, MSITEM.vmodi, MSITEM.dmodi "
 + " , MSITEM.nid_uom_sales, MSITEM.nid_uom_purchase ";
le += " FROM "
 + " tblm_item MSITEM "
 + " LEFT JOIN tblm_item PRITEM ON PRITEM.nid = MSITEM.nparent_id "
 + " LEFT JOIN tblm_setting MSSETT on MSSETT.vcode = MSITEM.vitem_type and MSSETT.vtype = 'item_type' "
 + " LEFT JOIN tblm_currency MSCURR on MSCURR.nid = MSITEM.nid_currency "
 + " LEFT JOIN tblm_uom MSUOM_STK on MSUOM_STK.nid = MSITEM.nid_uom_stock "
 + " LEFT JOIN tblm_uom MSUOM_SLS on MSUOM_SLS.nid = MSITEM.nid_uom_sales "
 + " LEFT JOIN tblm_uom MSUOM_PRCHSE on MSUOM_PRCHSE.nid = MSITEM.nid_uom_purchase ";
re += " WHERE 1=1 ";
```

Gambar 3.72. Get Data Item

Gambar 3.72 menunjukkan *query* SQL yang mengambil data item dari tabel `tblm_item` beserta informasi terkait seperti tipe item, parent item, UoM, mata uang, dan harga. Kolom-kolom yang dipilih meliputi **itemID, itemCode, itemTypeCode, parentID, parentCode, currencyDesc, nprice, nweight, nvolume, naging**, dan **nstatus**. Data UoM (**nid_uom_stock, nid_uom_sales, nid_uom_purchase**) diambil melalui `LEFT JOIN` dengan tabel `tblm_uom`, sedangkan mata uang diambil dari `tblm_currency`. Query ini menyatukan informasi item secara lengkap untuk tampilan monitoring item dalam aplikasi.

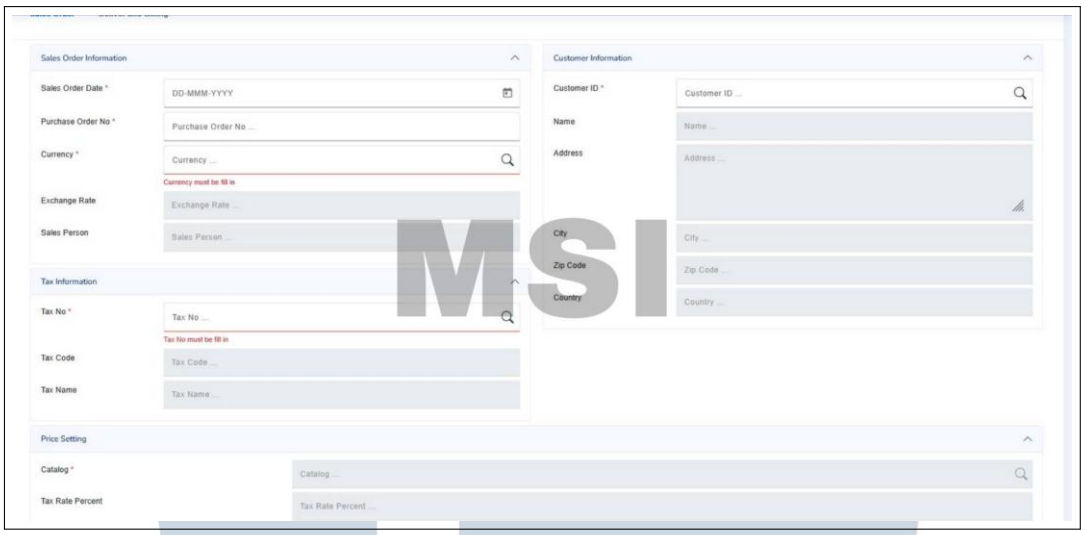
3.3.9 Membuat tampilan Sales Order & memperbaiki LoV

Pada bagian ini dibuat tampilan *sales order* sesuai dengan *mockup* yang ada. Dalam *sales order*, terdapat beberapa *field* yang wajib diisi sebagai ketentuan dari perusahaan. Berikut adalah bagain desain dari menu *Sales Order* untuk memperjelas bagaimana alur dari menu ini bekerja.



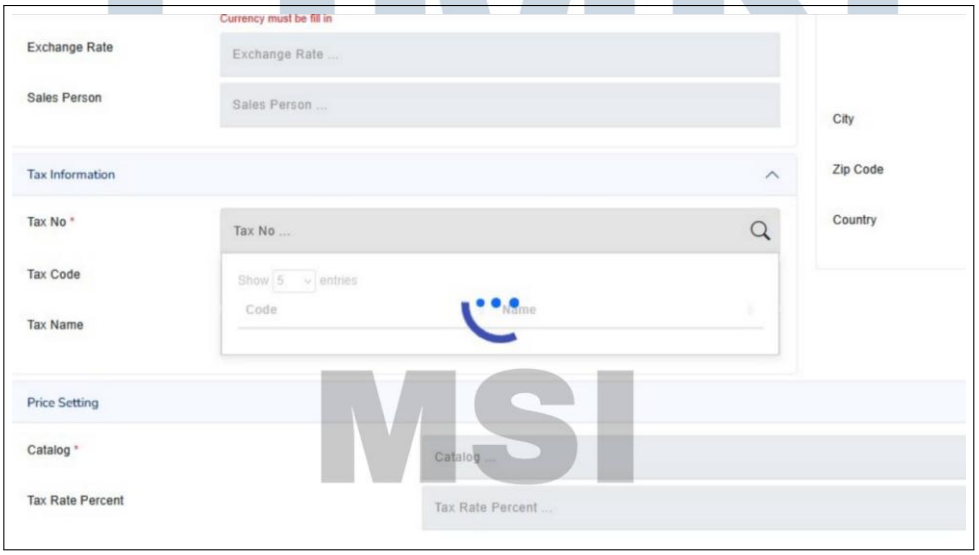
Gambar 3.73. Bagan Desain pada Menu Sales Order

Gambar 3.73 menampilkan *form* yang terdiri dari **Sales Order Information**, **Tax Information**, **Customer Information**, dan **Price Setting**. Pengguna harus mengisi *form* melalui pemilihan data dari komponen LoV. Ketika LoV telah berhasil dipilih, maka data terkait (*detail*) otomatis terisi pada *form*. Jika semuanya telah diisi, maka tabel Price Setting akan aktif dan pengguna dapat menambahkan item harga. Data *Sales Order* yang telah lengkap akan disimpan kembali ke *back-end*.



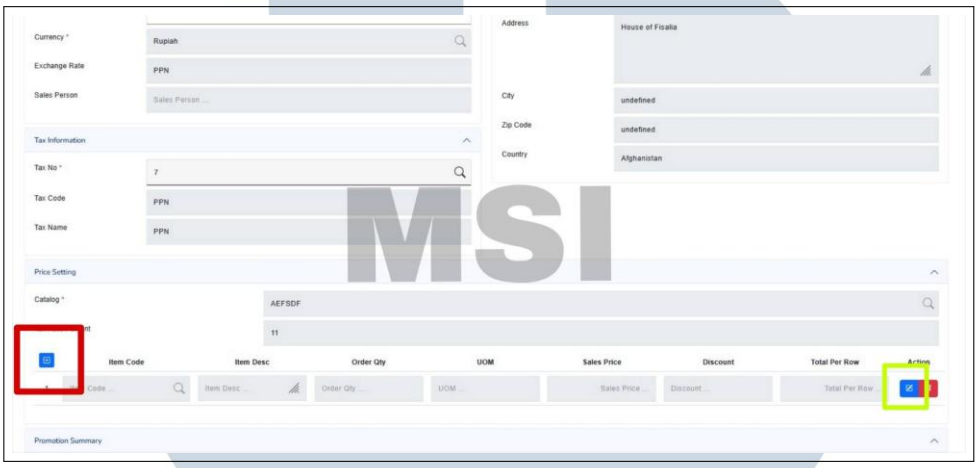
Gambar 3.74. Tampilan Sales Order

Tampilan *sales order* pada Gambar 3.74 mencakup beberapa informasi seperti **Sales Order Information, Tax Information, Customer Information**, dan **Price Setting**. Informasi *sales order* terdiri dari tanggal *sales order*, nomor *purchase order*, mata uang, *exchange rate*, dam *sales person*. Informasi pajak terdiri dari nomor pajak, kode, serta nama. Informasi pelanggan terdiri dari ID pelanggan, nama, alamat, kota, kode pos, dan negara. Khusus untuk informasi pelanggan dan informasi pajak, saat data pada LoV dipilih, maka data-data dibawahnya akan otomatis terisi sesuai ID. Kemudian ada pengaturan harga yang terdiri dari katalog produk dan tarif pajak persen.



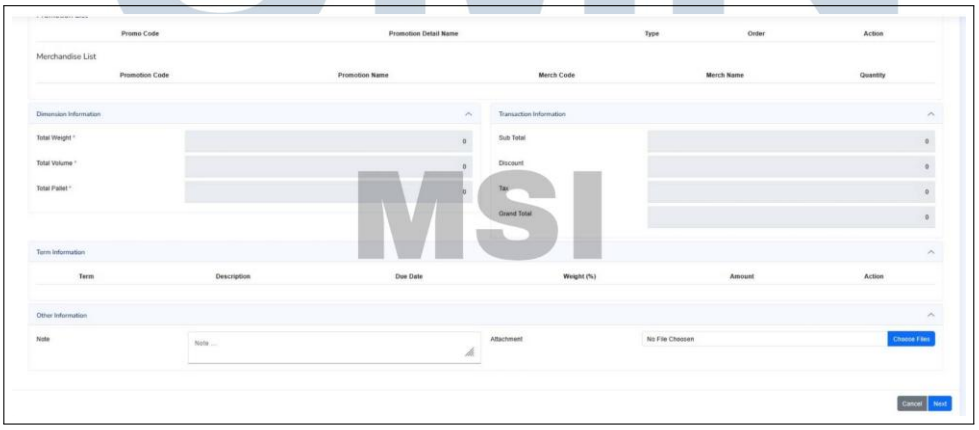
Gambar 3.75. Error pada LoV Sales Order

Pengaturan LoV Gambar 3.75 mengalami error pada awalnya, hal ini terjadi karena konektivitas yang salah pada *database* dan *query* yang tidak sesuai. Sehingga, LoV tidak dapat mengidentifikasi data yang ada dan terjadi *loading* secara terus-terusan seperti gambar tersebut.



Gambar 3.76. Perbaikan pada Sales Order

Gambar 3.76 menampilkan data-data yang telah berhasil dipilih dan dimunculkan. Dalam *sales order*, setiap item yang ingin ditambah pada pengaturan harga hanya dapat berjalan saat semua data pajak, pelanggan, dan katalog produk telah diisi, sehingga akan muncul tanda tambah seperti kotak merah di bagian kiri. Kotak hijau pada bagian kanan adalah tanda modifikasi yang saat diklik, maka akan mengaktifkan beberapa *field* untuk diisi. Jika telah diisi, maka tanda modifikasi akan berubah menjadi tanda centang. Tanda centang yang diklik akan menonaktifkan *field* sebelumnya.



Gambar 3.77. Perbaikan pada Sales Order

Gambar 3.77 adalah lanjutan dari tampilan *sales order* sebelumnya. Hanya saja fitur-fitur lain yang lebih kompleks akan dilanjutkan oleh developer yang lebih senior, hingga ke tahap pengumpulan data, serta pencetakan PDF. Dashboard *sales order* dibuat dengan perpaduan komponen global, sehingga tampilan-tampilan sejenis dapat digunakan kembali dan disesuaikan fungsinya sesuai kebutuhan.

3.4 Kendala dan Solusi yang Ditemukan

Saat pelaksanaan kerja magang, terdapat beberapa kendala yang ditemukan, yaitu:

1. Pada tahap awal pelaksanaan magang, terdapat kesulitan dalam menyesuaikan sintaks kode *boilerplate* perusahaan. Hal tersebut terjadi karena belum terbiasa dengan teknologi dan *framework* yang digunakan, terutama AngularJS dan Spring Boot.
2. Kesalahan dalam proses integrasi API antara *front-end* dan *back-end* sering terjadi.
3. Parameter suatu *method* kurang diperhatikan dengan baik, sehingga saat proses pengiriman data antara *front-end* dan *back-end*, sering terjadi kesalahan karena *back-end* tidak mampu mengidentifikasi nilai yang dikirim dari *front-end*. Akibatnya, sering terjadi **NullPointerException**.
4. Penggunaan aplikasi Netbeans untuk pengembangan *back-end* cukup memberatkan perangkat pribadi, sehingga proses menjalankannya membutuhkan waktu lebih lama. Akibatnya, fitur *built-in debugging* pada Spring Boot tidak dapat dijalankan dengan baik. Perlu diketahui untuk laptop yang digunakan adalah Lenovo Legion Y540 dengan spesifikasi prosesor Intel Core i5-9300H (4 Cores, 8 Threads, 2.4 GHz base clock, 4.1 GHz max turbo), GPU NVIDIA GeForce GTX 1650 (4GB GDDR5), RAM 8GB DDR4 2666MHz, dan storage 1TB SSD.
5. Struktur kode proyek yang sangat besar membuat proses identifikasi letak *error* menjadi lebih sulit. Proyek disebut besar karena memiliki 109 tabel dalam *database*, 747 *files* dan 34 *folders* yang mencakup *rest*, model, *view*, dao/dao *implementation*, *service/service implementation back-end*, lalu terdapat 622 *files* dan 160 *folders* pada *front-end* yang mencakup komponen-komponen global dan proyek utama. Kesulitan tersebut semakin meningkat

karena fitur *debugging* tidak dapat berfungsi dengan baik pada Netbeans di perangkat pribadi saat awal bekerja.

6. Tidak terbiasa dengan *query* SQL yang panjang dan kompleks, seperti yang terlihat pada Gambar 3.27 dan Gambar 3.72. Hal ini menyebabkan sering terjadi kesalahan penulisan *query* SQL, karena SQL ditulis dalam tanda kutip **(””)** sehingga jika terdapat kesalahan sintaks, IDE tidak dapat mendeteksinya kecuali aplikasi dijalankan terlebih dahulu.

Sehingga, dari beberapa kendala yang ditemukan saat bekerja, solusi yang telah ditemukan adalah:

1. Kesulitan dalam penyesuaian sintaks kode diatasi dengan melakukan pembelajaran intensif. Meskipun memerlukan waktu sekitar 2 hingga 3 hari untuk mempelajari alur kode, proses tersebut masih disertai banyak kesalahan saat menulis kode. Namun, seiring berjalannya waktu dan bimbingan dari senior, kemampuan penyesuaian terhadap sintaks kode dapat meningkat secara bertahap.

2. Kesalahan proses integrasi API diatasi dengan melakukan pengecekan struktur data secara menyeluruh, seperti memastikan tipe data dan format yang diharapkan oleh *back-end*. Selain itu, penambahan *logging* sangat membantu dalam melacak data dan *endpoint* yang dikirimkan, sehingga kesalahan dapat lebih mudah diidentifikasi.

3. Terjadinya **NullPointerException** dicegah dengan melakukan pengecekan pada awal metode menggunakan logika seperti `parameter != null` sebelum parameter tersebut digunakan dalam logika kode. Selain itu, penggunaan `console.log()` juga sangat efektif untuk melacak bagian kode di mana suatu metode kehilangan nilai pada parameternya.

4. Kendala terkait beratnya aplikasi Netbeans diatasi dengan memigrasikan proyek *back-end* dari Netbeans ke Visual Studio Code yang lebih cepat dan ringan.

5. Teratasinya masalah *debugging* pada *back-end*, kendala dalam menemukan letak *error* juga dapat diatasi. Alur eksekusi kode dapat dilacak setiap barisnya, sehingga proses identifikasi kesalahan menjadi lebih efektif.

6. Masalah penulisan *query* SQL diatasi dengan mempelajari ulang sintaks dasar dan kompleks SQL melalui dokumentasi *online*. Selain itu, penulisan *query* SQL dibagi menjadi beberapa bagian kecil agar lebih mudah dibaca dan dipahami. Untuk mendeteksi kesalahan *query*, metode *debugging* digunakan untuk memastikan apakah *query* berhasil dieksekusi. Alternatif lain adalah mengeksekusi *query* langsung melalui DBeaver untuk memverifikasi hasilnya sebelum diimplementasikan dalam kode.