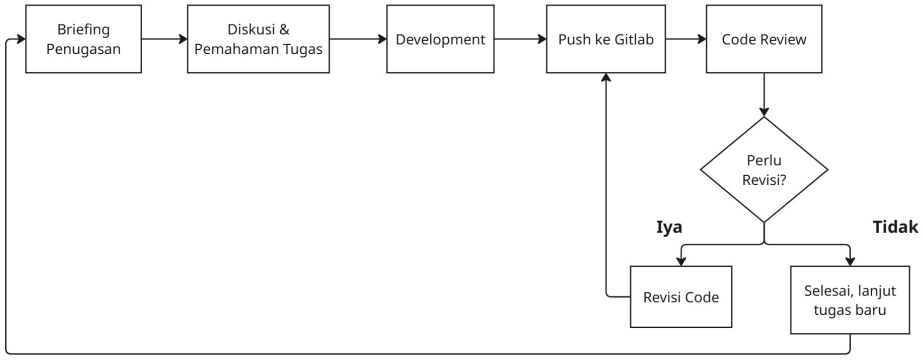
# BAB 3
# PELAKSANAAN KERJA MAGANG

## 3.1 Kedudukan dan Koordinasi

Selama magang di PT. Mitra Integrasi Digital, penulis berposisi sebagai *Junior Developer* yang berfokus pada pengembangan aplikasi android berbasis Kotlin dan Java. Dalam struktur organisasi, penulis bekerja di bawah arahan langsung bapak Boby Hartanto selaku supervisor sekaligus pemilik perusahaan dan bapak Boby Harmoko yang juga pemilik perusahaan serta ikut dalam pemberian arahan.




Gambar 3.1. Alur kerja magang di MID

Adapun alur pengerjaan tugas selama magang di MID, dimulai dari supervisor yang akan memberikan *briefing* penugasan yang dapat berupa pengembangan fitur maupun pemenuhan kebutuhan aplikasi Moni POS, lalu kemudian penulis melakukan diskusi dengan supervisor untuk memahami spesifikasi dan detail dari tugas yang diberikan. Setelah memahami dengan jelas, penulis melaksanakan pengembangan sesuai dengan arahan yang telah ditetapkan.

Setelah pengerjaan selesai, penulis melakukan push kode ke *repository* GitLab perusahaan untuk dilakukan proses *review*. Supervisor akan melakukan *code review* terhadap hasil pengerjaan, dan jika ditemukan hal yang perlu diperbaiki akan memberikan masukan atau arahan untuk melakukan revisi.

## 3.2 Tugas yang Dilakukan

Penulis bekerja di MID selama 4 bulan, dengan total 84 hari, 645 jam. Selama kerja magang di MID, penulis berfokus pada pengembangan fitur dan penyesuaian aplikasi Moni POS dengan backend terbaru yang sudah ada menggunakan Golang. Sebelumnya Moni POS terintegrasi dengan backend yang lama yaitu nodejs, namun karena beberapa dependensi sudah *deprecated*, akhirnya MID memutuskan untuk menggunakan backend baru yang mereka buat dengan Golang. Selama periode magang, supervisor memberikan penulis berbagai tugas pengembangan yang jika disimpulkan menjadi point point berikut:

1. Implementasi fitur *Pending QRIS* - Pengembangan sistem untuk menangani transaksi *QRIS* yang berstatus pending dari pihak bank.
2. Merubah tampilan antarmuka pada beberapa halaman - Penyesuaian tampilan Android dengan sistem backend Golang yang baru untuk menciptakan antarmuka yang lebih fresh dan modern.
3. Halaman *Printer Setting* dan Sentralisasi fungsi printer - Pembuatan halaman pengaturan printer dengan fitur pairing Bluetooth, konfigurasi karakter per-baris dan ukuran kertas struk, disertai sentralisasi fungsi print untuk arsitektur yang lebih modular dan mudah dipelihara.
4. Sistem Notifikasi Real-time Pesanan Online - Implementasi sistem penerimaan pesanan online dengan notifikasi *real-time* melalui komunikasi socket server dan protokol gRPC untuk penanganan *self-order* otomatis
5. Implementasi Option - Implementasi sistem option menu yang terintegrasi ke dalam *cart*, transaksi, dan semua jenis struk printer.

Untuk uraian per minggu-Nya dapat dilihat pada tabel 3.2 yang ada pada Sub Uraian Pelaksanaan Magang.

## 3.3 Uraian Pelaksanaan Magang

Pelaksanaan kerja magang diuraikan seperti pada Tabel berikut.

| Minggu Ke - | Pekerjaan yang dilakukan |
|---|---|
| 1 | Pengenalan perusahaan, setup environment, mempelajari GO dan mendalami kembali modul mobile programming, konsep rest API |
| 2-3 | Clone project Monipos, mempelajari arsitektur, struktur file, troubleshooting build error dan dependensi deprecated |
| 3-4 | Clone golang Backend, eksplor Golang backend dan flow pembayaran Monipos, Pengerjaan fitur pending QRIS |
| 5 | Menyesuaikan monipos dengan backend golang(baik handle data/endpoint), Revamp UI recap sales, implementasi auto print pada pending QRIS. |
| 6 | Fix layout struk, and testing semua print struk pada 3 jenis printer, revamp page today transaction. |
| 7-8 | Pembuatan page printer setting dengan fitur bluetooth pairing, scan device, connection management, CPL & paper size setting |
| 9-10 | Refactor print service menjadi modular dengan PrintService interface untuk semua printer |
| 10 | Finalisasi print system, redesign login, develop order online dengan Socket.IO dan gRPC |
| 11-12 | Implementasi socket server per-tenant, sistem status order real-time, develop item options pada menu dan integrasi ke transaksi dan cart |
| 13 | Database migration options, finalisasi option pricing, implementasi fitur batalkan transaksi |
| 14 | Display option di struk printer, filter by date pada list pending QRIS, handle ketika toko tutup |
| 15 | Fix pembayaran dengan wallet internal, implement redis caching pada backend, Add handle scan qr for android. |
| 16-17 | Implementasi fitur edit menu dengan access control tenantOwner untuk visibility management |
| 18 | Dipindahkan untuk membuat apps HRIS bernama CMOasik |

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

### 3.3.1 Pengenalan Perusahaan dan *Setup Environment Development*

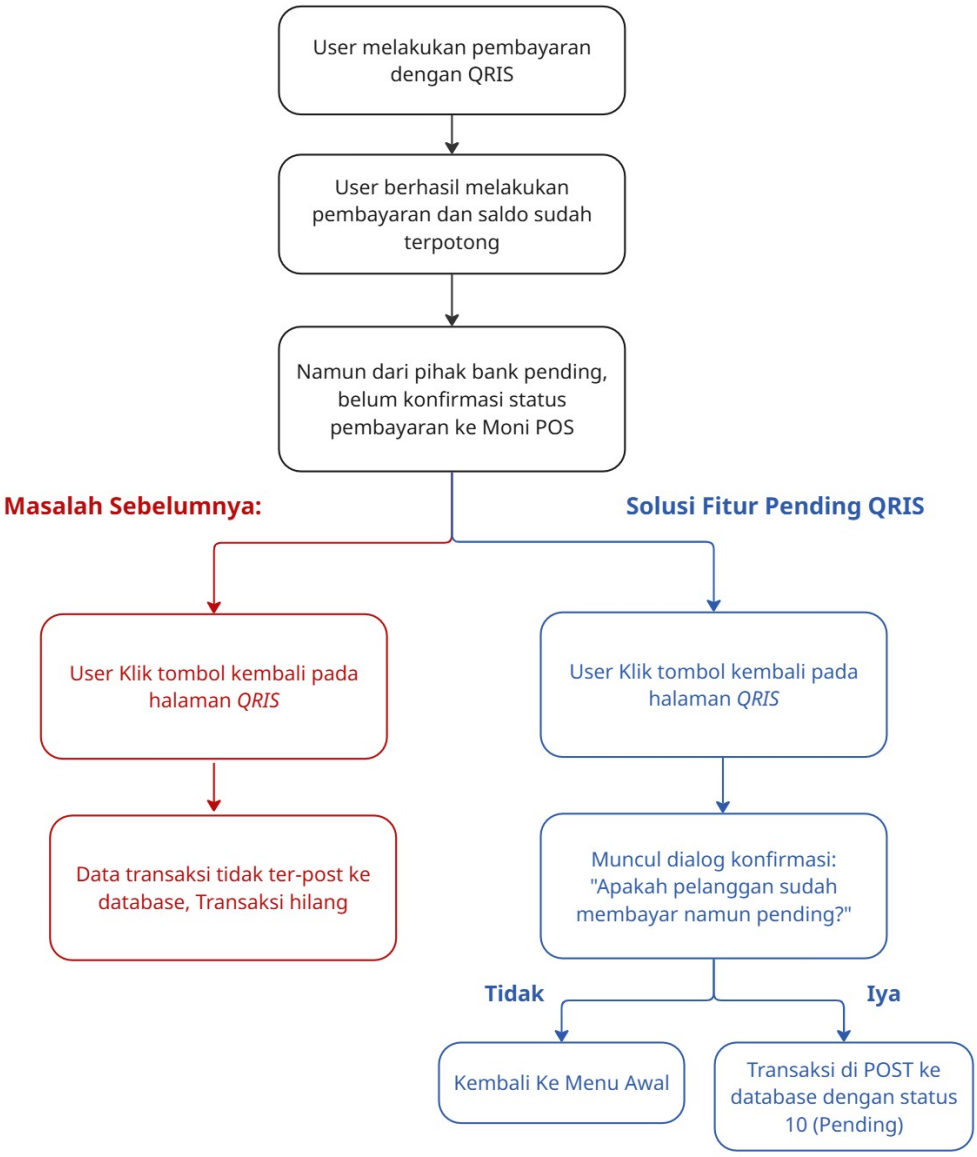
Pada hari Sabtu di minggu pertama, penulis bertemu langsung dengan kedua pemilik PT. Mitra Integrasi Digital di Mall Living World Alam Sutera. Pertemuan

ini dilakukan untuk melaksanakan interview sekaligus pengarahan mengenai tugas-tugas yang akan dikerjakan oleh penulis. Dalam pertemuan ini, kedua pemilik juga memberikan *briefing* serta arahan terkait fitur pertama yang akan dikembangkan, yaitu fitur *pending QRIS*. Lalu pada hari Senin (masih masuk kedalam minggu pertama), penulis dibuatkan akun *git* untuk dapat mengakses ke r*epository git* milik perusahaan. Di minggu ini juga penulis menginstall beberapa alat yang nantinya akan digunakan selama proses *development*, seperti Dbeaver untuk manajemen *database*, Redis Insight untuk melihat dan mendapatkan gambaran terkait data yang ada di dalam Redis, Wireguard untuk koneksi *VPN* ke server perusahaan, PostMan untuk pengujian *API*, dan install golang sebagai bahasa pemrograman *backend*. Di minggu ini penulis juga mempelajari Golang, dan mendalami kembali terkait modul *mobile programming* dan konsep *rest api*.

Memasuki minggu kedua penulis melakukan *clone repository* project android Moni POS dan mempelajari arsitektur serta struktur file-Nya. Di minggu ini juga penulis menyesuaikan versi *jdk* dan versi android studio untuk menyesuaikan dengan proyek. Lalu kemudian penulis menemukan masalah baru, yaitu proyek tidak dapat dijalankan akibat beberapa dependensi yang digunakan oleh project tersebut sudah *deprecated* dan tidak tersedia lagi di *Jcenter*. Oleh karena itu, penulis berusaha mencari dependensi pada directory yang masih aktif seperti Maven Central, Github, dan Jitpack.io. Semua dependensi yang bermasalah dapat diselesaikan, kecuali 1 dependensi yaitu sunmidslib yang tidak dapat ditemukan di *repository* manapun. Akhirnya, sebagai solusi sementara, penulis menonaktifkan bagian kode yang menggunakan dependensi tersebut agar proyek dapat dijalankan.

### 3.3.2 Pengerjaan Fitur *Pending QRIS*

Pengerjaan fitur *Pending QRIS* berlangsung selama 2 minggu, yaitu pada minggu ke-3 dan ke-4. Fitur ini bertujuan agar data transaksi yang pembayarannya sudah berhasil diproses namun statusnya masih pending karena masalah dari sisi bank, tetap tersimpan ke server ketika pengguna mengklik tombol kembali pada halaman pembayaran QRIS Moni POS.




Gambar 3.2. Flow solusi fitur Pending QRIS

Pada gambar 3.2 merupakan *flow* solusi fitur *Pending* QRIS yang menunjukkan perbandingan antara kondisi sebelumnya dengan solusi yang dikembangkan. Sebelum implementasi fitur ini, terdapat permasalahan yang cukup serius dalam sistem pembayaran QRIS dimana ketika *customer* telah berhasil melakukan pembayaran melalui aplikasi *e-wallet* atau *mobile banking* mereka, namun status pembayaran berhasil mengalami *delay* dari pihak bank untuk dikirim ke Moni POS. Dalam kondisi seperti ini, jika kasir menekan tombol *close* untuk membatalkan transaksi, maka seluruh data transaksi akan hilang dan tidak tersimpan ke server, sehingga menyebabkan kerugian bagi kedua belah pihak.

Dengan adanya fitur ini, pelanggan yang sudah berhasil melakukan pembayaran namun mengalami *delay* konfirmasi dari pihak bank, data transaksinya akan tetap tersimpan di *database* server dengan status *pending*. Hal ini mencegah kerugian finansial yang dialami oleh *merchant* maupun pelanggan, sekaligus mencegah hilangnya data transaksi yang penting untuk proses *audit* keuangan. Fitur ini juga memungkinkan tim operasional untuk melakukan *follow-up* dan verifikasi status pembayaran secara manual, sehingga transaksi yang sebenarnya sudah berhasil dapat diperbaharui statusnya menjadi *complete* di kemudian hari.

## A Komponen utama yang dikembangkan

Pada pengerjaannya, terdapat 3 komponen utama dalam pengembangan fitur *Pending* QRIS ini. Yang pertama Yaitu fungsi *Post Pending Transaction*, kedua adalah *PendingQrisActivity*, terakhir adalah *DetailPendingQrisActivity*. Untuk penjelasan lebih lengkapnya terdapat pada berikut:

### A.1 Fungsi *Post Pending Transaction*

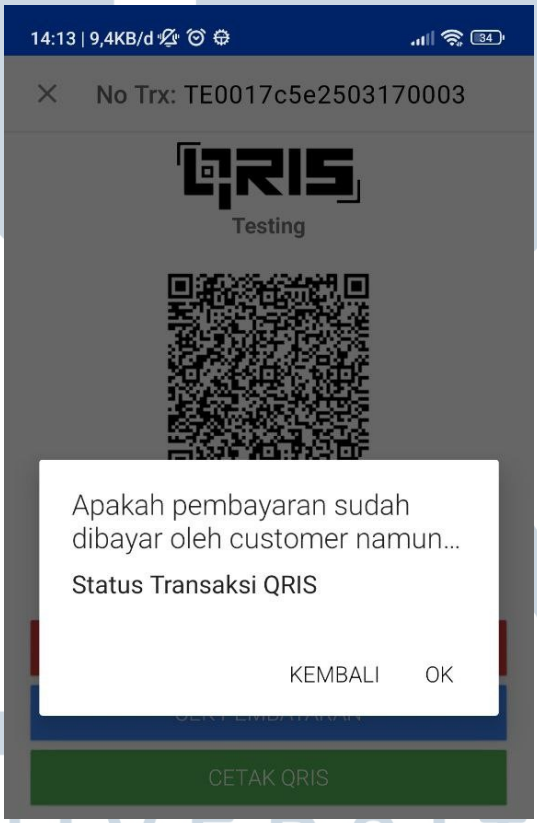
Fungsi `postPendingTransaction()` merupakan inti dari fitur *Pending QRIS* yang bertanggung jawab untuk melakukan proses *POST* dan menyimpan data transaksi ke database server dengan status transaksi 10 (*pending*). Fungsi ini berada di *QrisPayment dialog* dan akan dipanggil ketika user memberikan konfirmasi positif pada dialog yang muncul.

```
img_dialog_fullscreen_close = findViewById(R.id.img_dialog_fullscreen_close)
img_dialog_fullscreen_close.setOnClickListener(View.OnClickListener { v: View? ->
    if (!transtimeout) {
        // Confirmation Dialog
        MoniposApp.showQrisStatusConfirmDialog(
            mContext,
            title: "Status Transaksi QRIS",
            message: "Apakah pembayaran sudah dibayar oleh customer namun masih pending di sistem?"
        ) { dialog, which ->
            if (which == DialogInterface.BUTTON_POSITIVE) {
                // YES - POST pending transaction
                postPendingTransaction()
            } else {
                // NO - Cancel transaction
                dismiss()
            }
        }
```

Gambar 3.3. Potongan kode confirmation dialog pada QRIS payment dialog

Berdasarkan potongan kode pada gambar 3.3, dapat dilihat bahwa ketika

tombol *close* (`img_dialog_fullscreen_close`) ditekan, sistem akan mengecek kondisi `!transtimeout` terlebih dahulu. Jika kondisi tersebut terpenuhi, maka akan muncul dialog konfirmasi dengan judul ”Status Transaksi QRIS” dan pesan ”Apakah pembayaran sudah dibayar oleh *customer* namun masih *pending* di sistem?”. Ketika kasir memilih opsi ”Iya” yang ditandai dengan `DialogInterface.BUTTON_POSITIVE`, maka sistem akan mengeksekusi fungsi `postPendingTransaction()` untuk memproses dan menyimpan data transaksi dengan status *pending*. Sebaliknya, jika kasir memilih opsi ”Tidak”, maka sistem akan menjalankan fungsi `dismiss()` yang akan membatalkan transaksi dan menutup dialog.

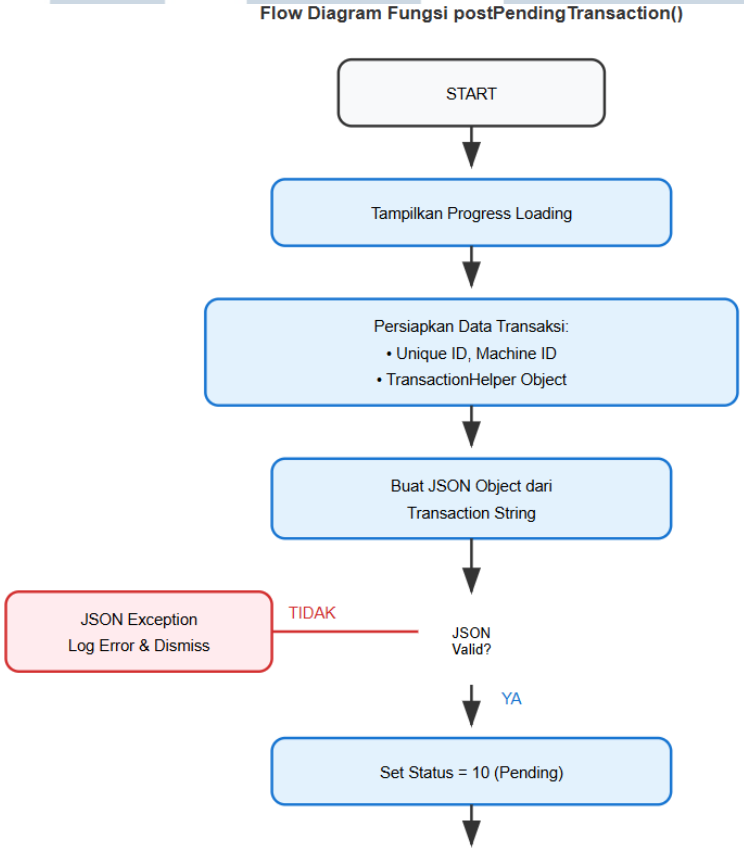



Gambar 3.4. Gambar Dialog konfirmasi Pada QRIS payment dialog

Gambar 3.4 menunjukkan implementasi visual dari dialog konfirmasi yang telah dijelaskan dalam kode sebelumnya. Pada dialog ini, kasir akan ditanyakan mengenai status pembayaran *customer*, dengan dua pilihan tombol yang berbeda fungsinya. Tombol ”OK” akan memicu eksekusi fungsi `postPendingTransaction()` yang akan menyimpan data transaksi ke server dengan status *pending*, sedangkan tombol ”KEMBALI” akan mengaktifkan fungsi

`dismiss()` yang membatalkan proses dan mengarahkan kembali ke menu utama. Desain *interface* yang sederhana ini memungkinkan kasir untuk dengan mudah memahami konsekuensi dari setiap pilihan yang diambil.

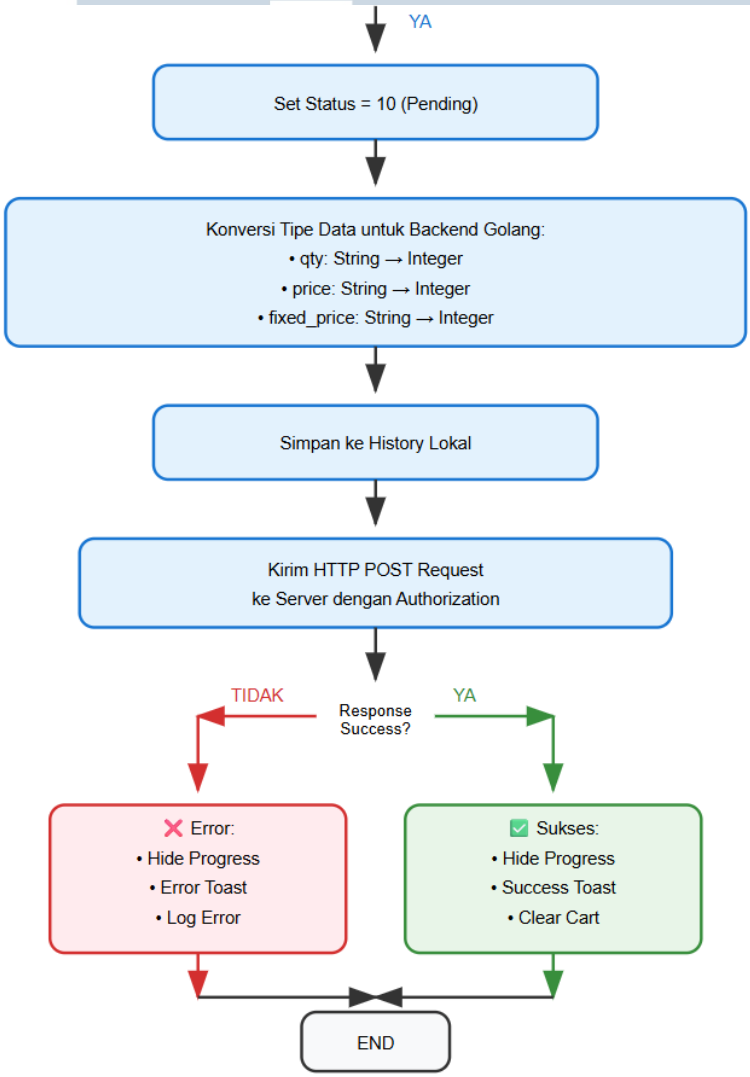
Setelah penjelasan mengenai dialog konfirmasi di atas, selanjutnya akan masuk kedalam penjelasan isi dari fungsi post pendingtransaction itu sendiri. Fungsi ini merupakan komponen inti yang bertanggung jawab untuk memproses dan menyimpan data transaksi dengan status pending ke server. Untuk memahami alur kerja fungsi ini secara detail, dapat dilihat pada flow diagram berikut yang menunjukkan tahapan-tahapan pemrosesan data transaksi.




Gambar 3.5. Flow Diagram Fungsi postPendingTransaction() - Bagian Persiapan Data

Pada gambar 3.5 menunjukkan fase awal dari fungsi `postPendingTransaction()` yang diawali dengan menampilkan *progress loading* untuk memberikan *feedback* visual kepada pengguna bahwa sistem sedang memproses transaksi. Tahap selanjutnya adalah persiapan data transaksi yang meliputi pengambilan *Unique ID* dan *Machine ID* untuk identifikasi transaksi, serta pembuatan *TransactionHelper Object* yang berfungsi sebagai *wrapper* untuk

semua data transaksi yang diperlukan. Setelah data transaksi disiapkan, sistem akan membuat *JSON Object* dari *transaction string* yang telah dihasilkan. Pada tahap ini terdapat pengecekan validitas JSON, dimana jika terjadi *JSON Exception*, sistem akan mencatat *error* ke *log* dan menghentikan proses dengan menutup dialog. Namun jika JSON berhasil dibuat, maka sistem akan melanjutkan ke tahap berikutnya yaitu mengatur status transaksi menjadi 10 yang menandakan status *pending*.




Gambar 3.6. Flow Diagram Fungsi postPendingTransaction() - Bagian Pemrosesan dan Pengiriman Data

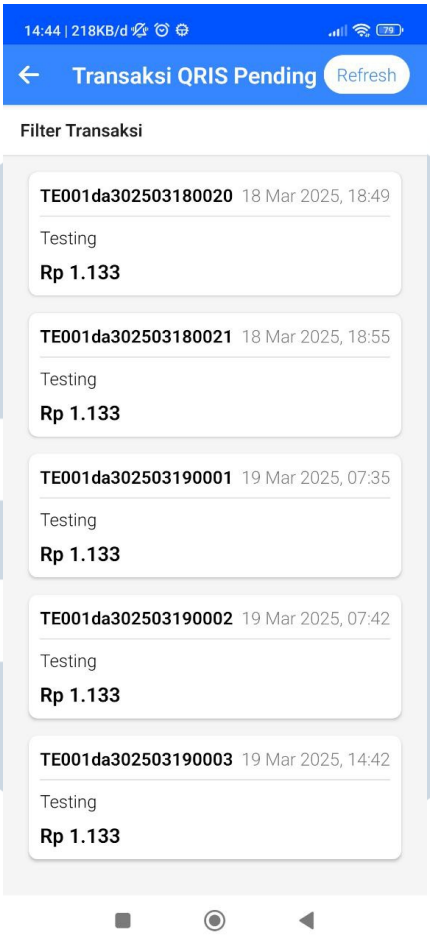
Gambar 3.6 adalah kelanjutan dari proses fungsi `postPendingTransaction()` setelah status transaksi berhasil diatur menjadi 10 (*pending*). Tahap selanjutnya adalah konversi tipe data untuk memastikan

kompatibilitas dengan *backend* Golang, dimana sistem akan mengubah tipe data *qty*, *price*, dan *fixed price* dari *string* menjadi *integer* sesuai dengan format *struct* yang diharapkan oleh server. Setelah konversi data selesai, sistem akan menyimpan data transaksi ke *history* lokal sebagai *backup*, kemudian membuat *HTTP POST request* yang dilengkapi dengan *header Authorization*. Pada tahap pengiriman data, terdapat dua kemungkinan respons, yaitu jika *request* gagal, sistem akan menjalankan *error handling* yang meliputi menyembunyikan *progress indicator*, menampilkan *error toast* kepada pengguna, mencatat *error* ke *log*, dan menutup dialog. Sebaliknya, jika *request* berhasil, sistem akan menjalankan *success handling* yang meliputi menyembunyikan *progress indicator*, menampilkan *success toast* dengan pesan ”Transaksi berhasil disimpan dengan status PENDING”, membersihkan data *cart*, dan menutup dialog. Kedua alur tersebut akan berakhir pada titik END yang menandakan selesainya eksekusi fungsi.

### A.2 *Activity Pending* QRIS (PendingQrisActivity)

Halaman ini dibuat untuk menampilkan daftar semua transaksi yang berstatus *pending* sebagai bagian dari sistem manajemen transaksi yang belum terselesaikan. *Activity* ini berfungsi sebagai halaman khusus yang memungkinkan kasir untuk mengelola transaksi-transaksi yang telah disimpan dengan status 10 (*pending*) melalui fitur *Pending QRIS* yang telah dijelaskan sebelumnya. Di dalamnya akan melakukan pemanggilan *API* ke *backend* untuk mengambil dan menampilkan data transaksi yang berstatus *pending*, sehingga memudahkan proses *follow-up* dan verifikasi pembayaran yang tertunda. *Activity* ini juga dilengkapi dengan fitur refresh untuk memperbarui data secara real-time dan filter transaksi untuk memudahkan pencarian data spesifik.

Setiap item transaksi dalam daftar menampilkan informasi penting seperti nomor transaksi, waktu transaksi, nominal pembayaran, dan keterangan produk yang dibeli. Interface yang *user-friendly* memudahkan kasir untuk mengidentifikasi transaksi mana yang perlu ditindaklanjuti berdasarkan waktu dan nominal transaksi. Dan juga adanya fitur *swipe refresh* yang memudahkan pengguna untuk mendapatkan data yang selalu *up-to-date* dengan server. Berikut tampilan halaman dari *PendingQrisActivity*:



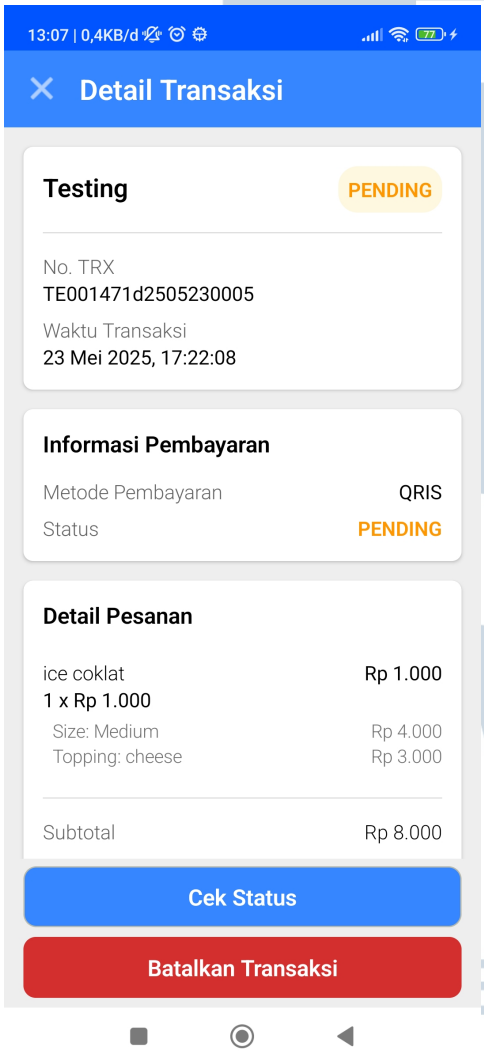

Gambar 3.7. Halaman *List Pending Qris*

## A.3 Activity Detail Pending QRIS (DetailPendingQrisActivity)

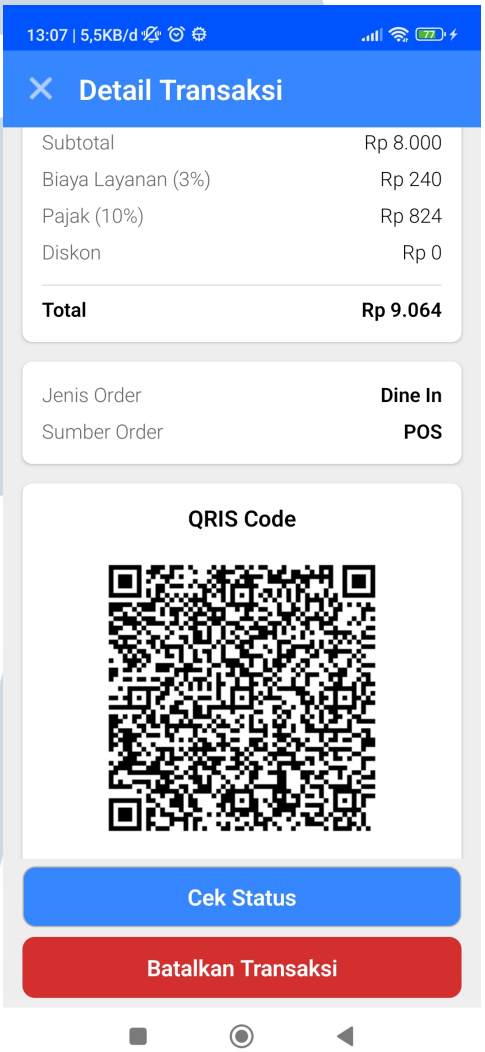
Pada halaman ini, terdapat 2 aksi yang dapat dilakukan oleh *user*. Pertama adalah mengecek status. Pada sisi *android*, untuk mengecek status hanya tinggal memberikan *id-transaksi merchant* ke *API* server milik MID. Server MID akan mengembalikan *response* dalam format JSON yang berisi informasi status pembayaran. *Response* tersebut memiliki dua kolom utama yaitu pesan yang menunjukkan status umum *request* dan *trxInfo.status* yang berisi kode status spesifik pembayaran. Jika kolom pesan bernilai ”*success*” dan *trxInfo.status* bernilai ”02”, maka pembayaran dinyatakan berhasil dan sistem akan otomatis memanggil fungsi *updateTransactionStatus()* untuk mengubah status transaksi menjadi *success* (status 8) di *database backend*. jika data status pada respon belum bernilai angka 2, artinya pembayaran masih *pending* dari pihak bank.

Pada fungsi *updateTransactionStatus*, fungsi tersebut akan *request* PATCH

ke *backend API* golang untuk mengubah status menjadi 8. Aksi ke dua adalah membatalkan transaksi. *User* dapat membatalkan transaksi dengan menekan tombol batalkan transaksi. Nantinya akan menampilkan konfirmasi dialog apakah *user* benar benar ingin membatalkan transaksi ini, jika iya maka sistem akan memanggil fungsi *cancelTransaction* dan fungsi tersebut akan *request* PATCH ke *backend API* golang untuk mengubah status transaksi menjadi 9. Dan berikut untuk tampilan halaman *DetailPendingQris*:




Gambar 3.8. Halaman Detail *Pending* Qris-1.1




Gambar 3.9. Halaman Detail *Pending* Qris-1.2

### 3.3.3 Pengerjaan halaman Pengaturan Printer dan Sentralisasi fungsi printer

#### A Arsitektur Sentralisasi Print Service

Dalam pembuatan sentralisasi fungsi print, penulis menggunakan pola *Singleton Pattern* dan *Factory Pattern* untuk menciptakan sistem print yang modular dan dapat digunakan di seluruh aplikasi. Menurut Gang of Four [5], *Singleton Pattern* adalah ”pola desain perangkat lunak yang membatasi instansiasi sebuah kelas menjadi satu instance tunggal”. Sedangkan *Factory Pattern* menurut Fowler et al [6]., adalah ”pola yang mendefinisikan antarmuka untuk membuat objek, tetapi membiarkan subkelas memutuskan kelas mana yang akan diinstansiasi”. Arsitektur ini terdiri dari beberapa komponen utama:

#### A.1 *PrintService (Singleton)*

*PrintService* merupakan *central controller* yang mengelola seluruh operasi printer dalam aplikasi menggunakan *Singleton Pattern*. Sebagai *single point of access*, *PrintService* memastikan bahwa hanya ada satu *instance* yang menangani semua fungsi printing di seluruh aplikasi, sehingga mencegah konflik dan inkonsistensi. Ketika *Activity* atau komponen lain ingin menggunakan fungsi print, mereka harus memanggil *PrintService.getInstance(context)* disertai dengan data yang sesuai dengan model yang telah ditentukan untuk jenis print yang diinginkan. *PrintService* akan secara otomatis menentukan jenis printer yang tepat berdasarkan device model dan konfigurasi yang tersimpan.

```
/**
 * Print transaction history details
 */
fun printTransactionHistory(
    transaction: JsonObject,
    details: JsonArray,
    payments: JsonArray
) {
    try {
        reloadSettings()
        Log.d(TAG, msg: "Printing transaction history with settings: paperSize=${printerConfig.paperSize}, CPL=${printerConfig.charactersPerLine}")

        // Create data model for transaction history
        val historyData = HistoryData(
            transaction = transaction,
            details = details,
            payments = payments
        )

        // Get printer and print
        val printer = getPrinter()
        printer.printTransactionHistory(historyData)

    } catch (e: Exception) {
        Log.e(TAG, msg: "Error printing transaction history", e)
    }
}
```

Gambar 3.10. contoh fungsi print histori transaksi

Pada gambar 3.10 di atas dapat dilihat implementasi *method* `printTransactionHistory()` dalam PrintService yang menunjukkan pola kerja *central controller*. *Method* ini pertama-tama melakukan `reloadSettings()` untuk memastikan konfigurasi printer terbaru, kemudian mencatat *log* dengan detail pengaturan `paperSize` dan `charactersPerLine` yang sedang digunakan. Selanjutnya, sistem membuat *object* `HistoryData` dengan meng-*wrap* parameter input (`transaction`, `details`, `payments`) ke dalam struktur data yang standar. Langkah terakhir adalah memanggil `getPrinter()` yang menggunakan *Factory Pattern* untuk menentukan implementasi printer yang sesuai, lalu mengeksekusi *method* `printTransactionHistory(historyData)`. Pola ini memastikan bahwa setiap operasi *printing* selalu menggunakan konfigurasi terbaru dan printer yang tepat tanpa *Activity* perlu mengetahui detail implementasinya.

Didalam print *Service* ini juga terdapat pola Factory pattern yaitu pada bagian get printer() yang akan menentukan jenis printer berdasarkan karakteristik device.

```
private fun getPrinter(): Printer {
    val deviceModel = Build.MODEL.toUpperCase()

    return when {
        // For Sunmi devices with AIDL
        AidlUtil.getInstance().isConnect() && !deviceModel.startsWith( prefix: "O1") && !deviceModel.startsWith( prefix: "POS-OS01") ->
            SunmiPrinter(context, printerConfig, loginJson, tax, serviceTax)

        // For Verifone devices
        deviceModel.startsWith( prefix: "X990") ->
            VerifonePrinter(context, printerConfig, loginJson, tax, serviceTax)

        // Default bluetooth printer
        else ->
            return BluetoothPrinter(context, printerConfig, loginJson, tax, serviceTax)
    }
}
```

Gambar 3.11. fungsi getPrinter

*Method* `getPrinter()` mengimplementasikan *Factory Pattern* dengan melakukan deteksi otomatis model perangkat menggunakan `Build.MODEL.toUpperCase()`. Logika pemilihan printer menggunakan struktur `when` dengan tiga kondisi: jika perangkat mendukung Sunmi AIDL dan bukan model ”O1” atau ”POS-OS01”, maka akan mengembalikan `SunmiPrinter`; jika model perangkat dimulai dengan ”X990”, akan mengembalikan `VerifonePrinter`; dan untuk semua perangkat lainnya menggunakan `BluetoothPrinter` sebagai *default*. Setiap *instance* printer menerima parameter yang sama untuk memastikan konsistensi konfigurasi di semua implementasi.

## A.2 Printer *Interface*

*Interface* Printer adalah *blueprint* standar yang harus diimplementasikan oleh semua jenis printer, hal ini untuk memastikan konsistensi *method signature* dan *behavior*. *Interface* ini menggunakan prinsip *contract-based programming* dimana setiap implementasi printer (SunmiPrinter, VerifonePrinter, BluetoothPrinter) wajib menyediakan implementasi untuk semua *method* yang didefinisikan. Dengan adanya *interface* ini, PrintService dapat memanggil *method* yang sama tanpa perlu mengetahui detail implementasi spesifik dari masing-masing jenis printer. Berikut *interface* printer yang ada:

```kotlin
/**
 * Printer interface
 */
interface Printer {
    /**
     * Print transaction receipt
     */
    fun printTransaction(data: TransactionData)

    /**
     * Print sales recap
     */
    fun printSalesRecap(data: RecapData)

    /**
     * Print transaction history
     */
    fun printTransactionHistory(data: HistoryData)

    /**
     * Print pending QRIS receipt
     */
    fun printPendingQris(data: PendingQrisData)
    /**
     * Print QRIS code
     */
    fun printQris(data: QrisData)
}
```

Gambar 3.12. Interface printer

*Interface* Printer mendefinisikan lima *method* utama yang mencakup semua kebutuhan *printing* dalam aplikasi Moni POS. Setiap *method* menerima parameter berupa *data class* yang spesifik untuk jenis *print* tertentu, seperti `TransactionData` untuk struk transaksi normal, `PendingQrisData` untuk transaksi pending, dan `QrisData` untuk mencetak kode QR. Struktur ini memastikan *type safety* dan memudahkan *maintenance* karena setiap perubahan pada *interface* akan otomatis memaksa semua implementasi untuk menyesuaikan. Dengan pola ini, penambahan

jenis printer baru atau *method* baru dapat dilakukan dengan mudah tanpa mempengaruhi kode yang sudah ada. Dan dalam sistem ini menggunakan data classes seperti berikut salah satunya:

```kotlin
/**
 * Data class for transaction
 */
data class TransactionData(
    val transaction: com.google.gson.JsonObject,
    val details: com.google.gson.JsonArray,
    val payments: com.google.gson.JsonArray
)

/**
 * Data class for sales recap
 */
data class RecapData(
    val tobeprinted: com.google.gson.JsonObject,
    val details: com.google.gson.JsonArray,
    val payments: com.google.gson.JsonArray,
    val timeStart: String,
    val timeEnd: String
)
```

Gambar 3.13. Salah satu contoh data class

*Data classes* ini berfungsi sebagai *wrapper* untuk mengirimkan data yang diperlukan ke implementasi printer dengan struktur yang konsisten dan *type-safe*. `TransactionData` menggunakan struktur standar dengan `JsonObject` untuk data transaksi utama dan `JsonArray` untuk detail item serta informasi pembayaran. Sedangkan `RecapData` memiliki tambahan parameter `timeStart` dan `timeEnd` untuk menentukan periode laporan.

## A.3 Printer *Config*

Printer *config* digunakan untuk mengelola konfigurasi printer yang dapat disesuaikan user secara dinamis. Di dalamnya terdapat load printer setting yang digunakan untuk memuat pengaturan printer sebelum melakukan print. Juga terdapat *setPrinterConfig* untuk menyimpan perubahan pengaturan printer yang dilakukan oleh user.

```
fun loadPrinterSettings() {

    paperSize = 58

    charactersPerLine = 32
    Log.d(TAG,  msg: "Device model: $deviceModel")
    // Override with saved settings if available
    val savedPaperSize = Prefs.with(context).readString( key: "receipt_paper_size",  defaultString: null)
    savedPaperSize?.let {
        try {
            // Removes any non-digit characters from the string
            val numericValue = it.replace("[^0-9]".toRegex(),  replacement: "")
            // Convert the cleaned numeric string to an integer, assign to paperSize
            paperSize = numericValue.toInt()
        } catch (e: NumberFormatException) {
            Log.e(TAG,  msg: "Error parsing paperSize: $it", e)
        }
    }
    charactersPerLine = Prefs.with(context).readInt( key: "receipt_characters_per_line", charactersPerLine)
    Log.d(TAG,  msg: "Loaded printer settings: paperSize=$paperSize, charactersPerLine=$charactersPerLine")
}

fun setPrinterConfig(paperSize: Int, charactersPerLine: Int) {
    this.paperSize = paperSize
    this.charactersPerLine = charactersPerLine

    // Save to preferences
    Prefs.with(context).writeString("receipt_paper_size", paperSize.toString())
    Prefs.with(context).writeInt("receipt_characters_per_line", charactersPerLine)

    Log.d(TAG,  msg: "Printer config updated: paperSize=$paperSize, CPL=$charactersPerLine")
}
```

Gambar 3.14. Printer config

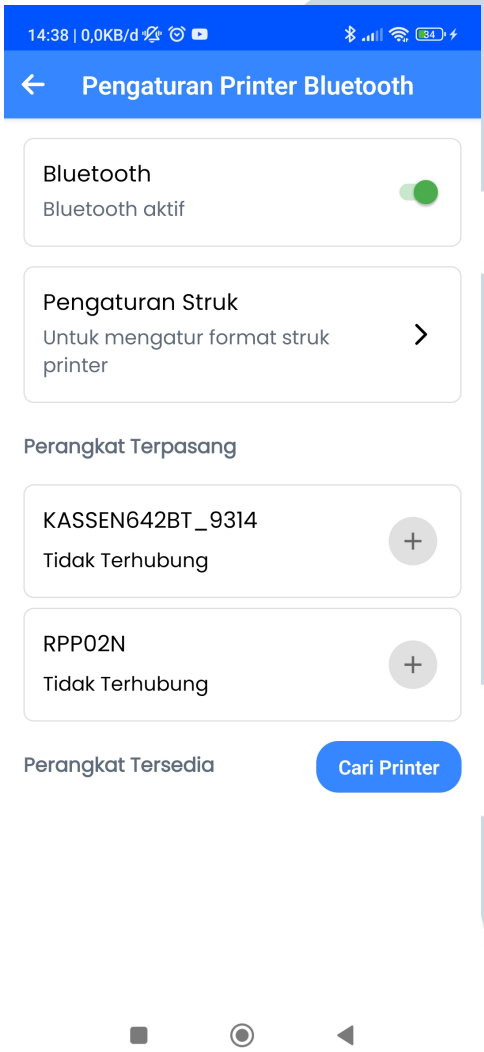
### A.4 File implementasi pada setiap printer

Terakhir, terdapat tiga file implementasi printer yang digunakan untuk mencetak pada masing-masing jenis printer, yaitu BluetoothPrinter, SunmiPrinter, dan VerifonePrinter. Setiap file memiliki lima fungsi print yang sama: print transaksi, print history transaksi, print QRIS, print rekap penjualan, dan print transaksi pending. Setiap file memiliki library masing masing untuk berkomunikasi dengan printer dan mencetak apa yang diinginkan. Untuk perangkat bluetooth menggunakan library DantSu EscPosPrinter, untuk perangkat sunmi menggunakan *AIDL (Android Interface Definition Language)*, dan untuk verifone menggunakan verifoneUtil.

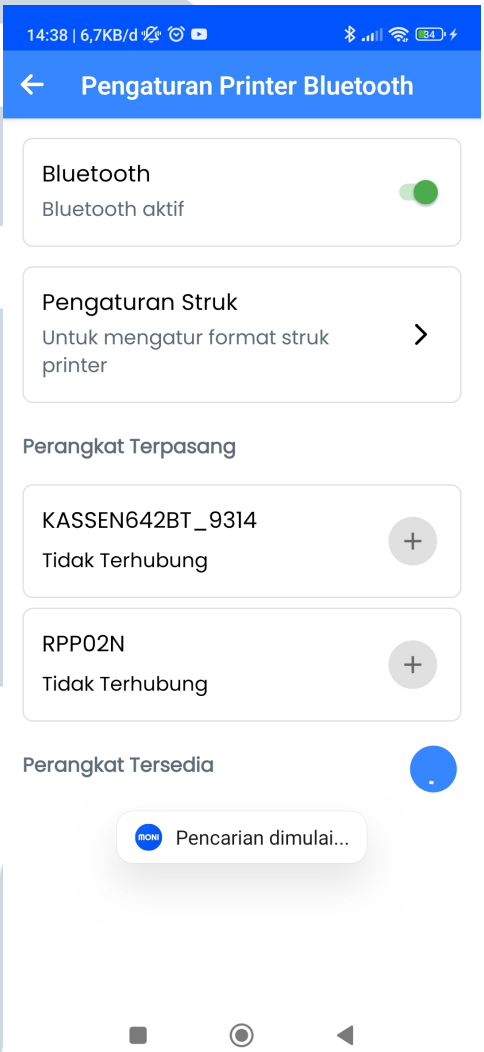
## B Halaman Pengaturan Printer

Halaman ini digunakan oleh user untuk dapat melakukan pencarian perangkat bluetooth printer yang ada di sekitar, melakukan pairing terhadap perangkat tersebut, dan menghubungkan ke perangkat tersebut. Setiap memasuki halaman ini, akan ada permintaan untuk mengaktifkan bluetooth apabila user blom mengaktifkan bluetoothnya. Di halaman ini nantinya terdapat tombol untuk

menuju halaman pengaturan struk. Di halaman pengaturan struk tersebut terdapat 2 komponen yang dapat diubah yaitu karakter per-baris dan ukuran kertas struk. Berikut contoh halamananya:

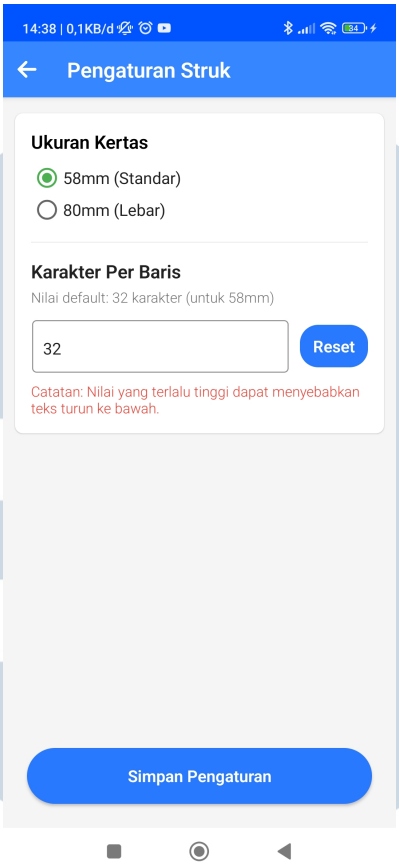



Gambar 3.15. Halaman pengaturan printer




Gambar 3.16. Mencari perangkat disekitar

Gambar 3.15 menunjukkan halaman utama pengaturan printer dengan fitur *toggle bluetooth*, menu pengaturan struk, dan daftar perangkat yang sudah terpasang maupun tersedia. Pada gambar 3.16, terlihat proses pencarian perangkat *bluetooth* aktif dengan tombol ”Cari Printer” yang memungkinkan sistem melakukan *scanning* perangkat di sekitar.




Gambar 3.17. Halaman pengaturan struk

Halaman pengaturan struk pada gambar 3.17 memungkinkan *user* untuk mengkonfigurasi karakter per-baris dan spesifikasi kertas yang digunakan. Terdapat dua pilihan ukuran kertas yaitu 58mm (standar) dan 80mm (lebar). Pengaturan karakter per baris dapat disesuaikan melalui *input field* dengan nilai *default* 32 karakter untuk kertas 58mm, dan tersedia tombol ”Reset” untuk mengembalikan ke pengaturan awal.

### 3.3.4 Merubah tampilan antarmuka pada beberapa halaman

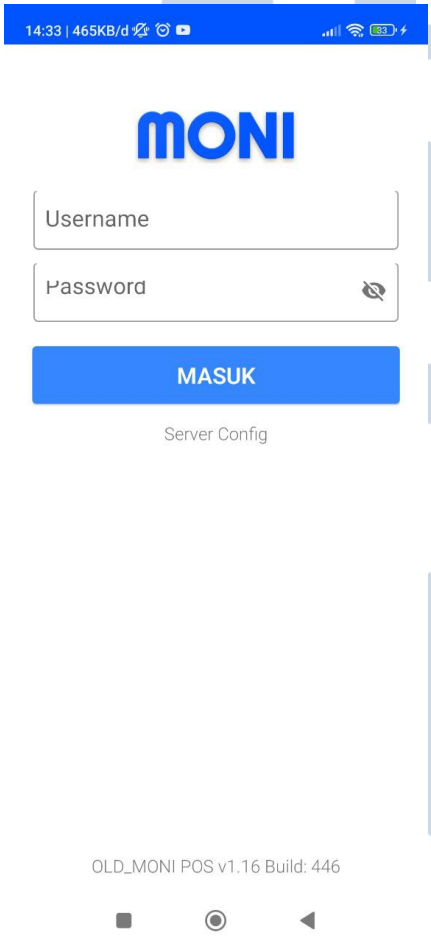
Saat magang kerja di MID, penulis mendapatkan tugas untuk merubah tampilan antarmuka pada beberapa halaman agar tampilan sedikit lebih fresh. Halaman yang perlu diubah oleh penulis diantaranya ialah:

1. Halaman Login
2. Halaman Rekap penjualan

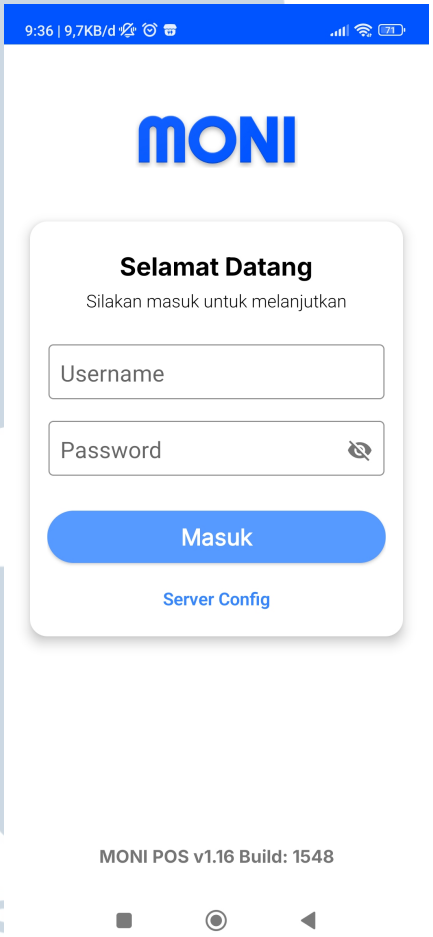
3. Halaman Keranjang
4. Halaman Pencarian transaksi
5. Halaman Transaksi hari ini

Dan berikut hasil sebelum dan sesudah perubahan pada setiap halaman:

## A Halaman Login



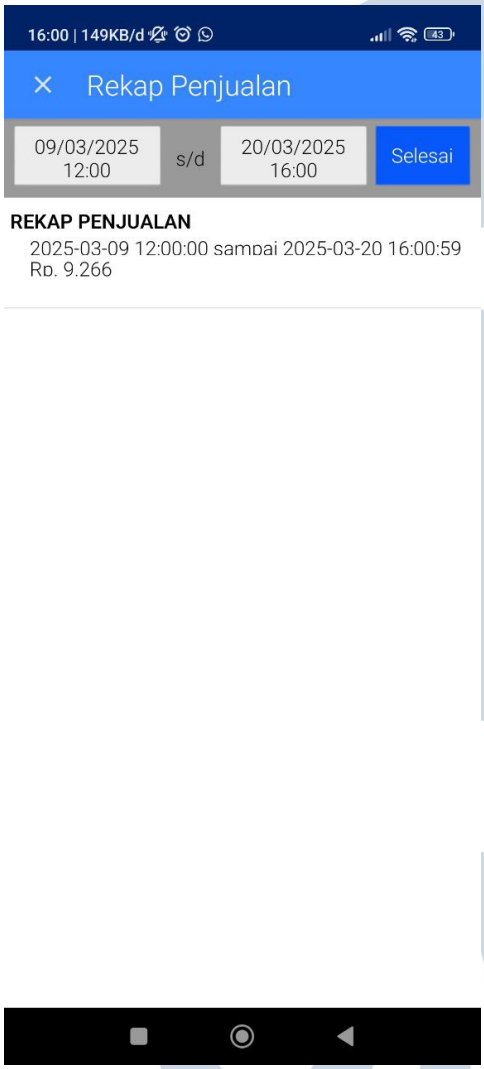

Gambar 3.18. Halaman login versi lama




Gambar 3.19. Halaman login versi terbaru

Pada gambar 3.18 merupakan tampilan halaman sebelum dilakukan desain ulang. Tampak hint dari kolom password *ter-overlap* oleh object lain. Kemudian pada gambar 3.19 merupakan tampilan halaman yang sudah diperbarui. halaman menggunakan *card-based design* dengan teks sambutan ”Selamat Datang” dan informasi versi aplikasi yang ditampilkan secara eksplisit.

## B Halaman Rekap penjualan

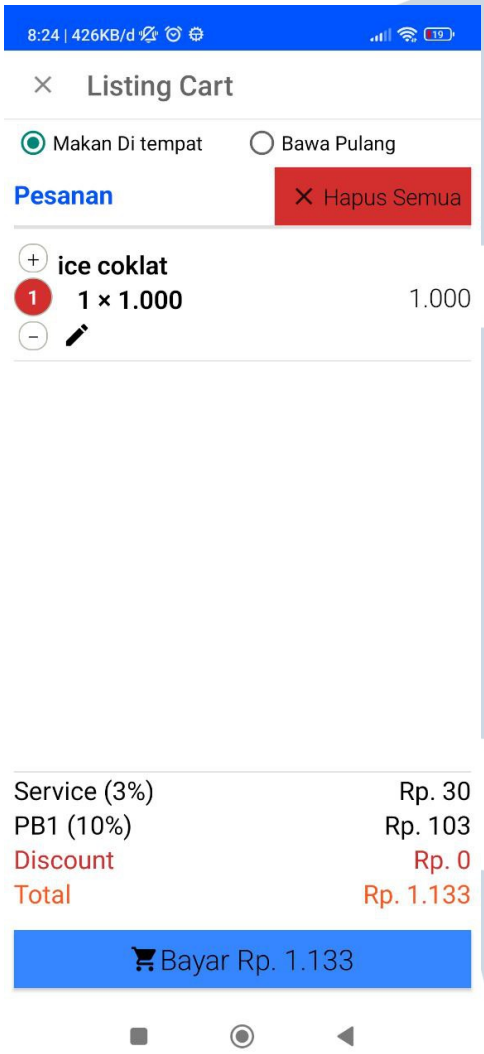



Gambar 3.20. Halaman recap versi lama

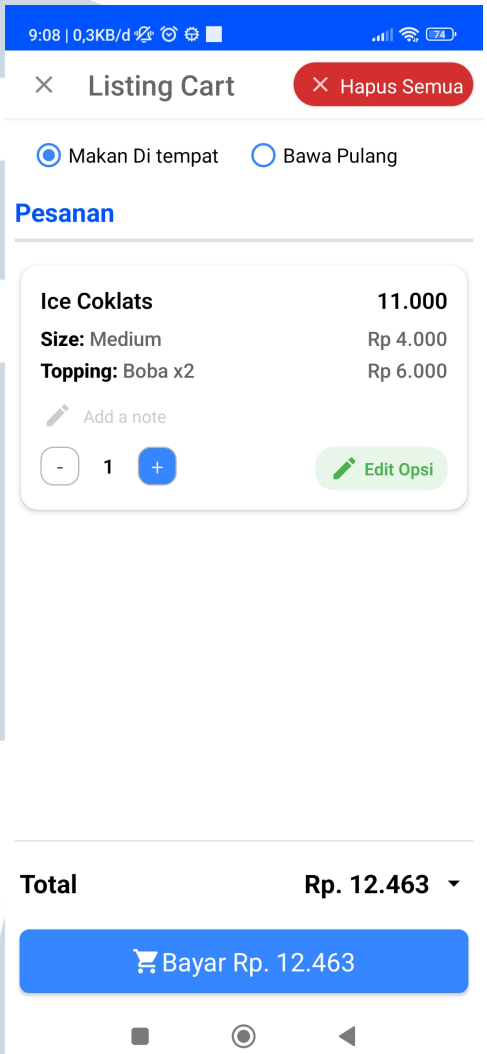

Gambar 3.21. Halaman recap versi terbaru

Pada lampiran 3.20 adalah halaman rekap penjualan versi lama. Lalu kemudian pada lampiran 3.21 merupakan versi halaman dari rekap penjualan yang telah dilakukan desain ulang. Halaman versi baru, menggunakan *date range picker* yang lebih *user-friendly* dengan ikon panah yang menunjukkan rentang periode, serta menampilkan hasil rekap dalam *card design* dengan *accent color* biru pada sisi kiri dan tombol ”Cetak” yang lebih menonjol.

## C Halaman keranjang




Gambar 3.22. Halaman keranjang versi lama




Gambar 3.23. Halaman keranjang versi terbaru

Pada tampilan versi lama (gambar 3.22), item ditampilkan dalam format sederhana dengan kontrol jumlah yang kurang menonjol dan detail pembayaran yang dicampur dengan item pesanan. Pada halaman baru yang terdapat pada lampiran 3.23, menggunakan *card-based design* untuk setiap item dengan informasi yang lebih terstruktur, dan menampilkan detail opsi yang dipilih. Perubahan ini membuat halaman keranjang lebih mudah dibaca dan digunakan, dengan kontrol yang lebih intuitif untuk pengguna.

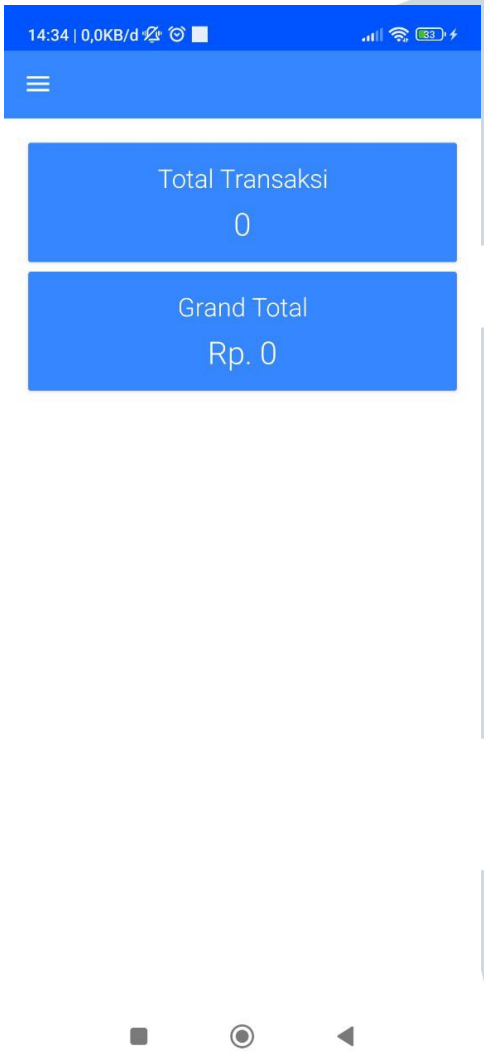
## D Halaman Pencarian transaksi


Gambar 3.24. Halaman Pencarian transaksi versi lama

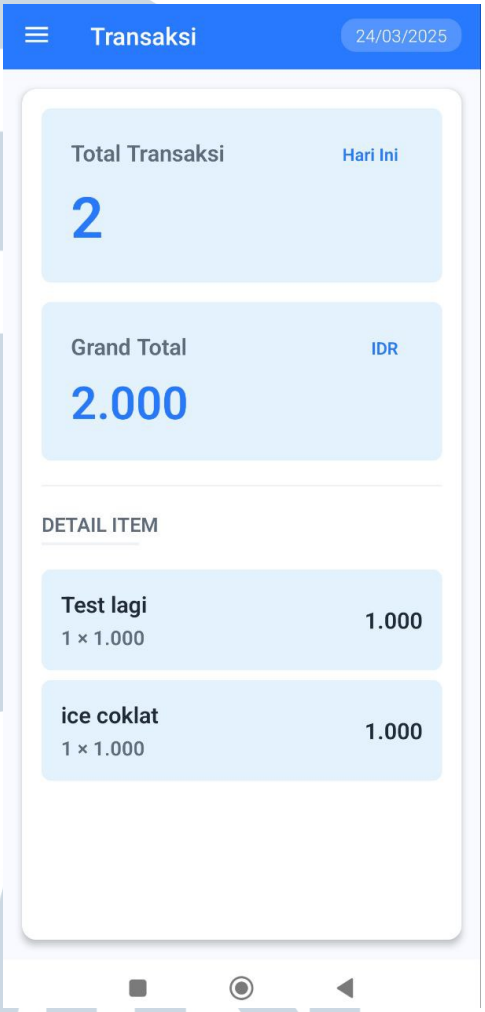

Gambar 3.25. Halaman Pencarian transaksi versi terbaru

Gambar 3.24 merupakan gambar dari halaman riwayat transaksi versi lama. Tampilan dari list data masih gabung menjadi satu dan tidak ada space atau margin antar list data. Kemudian pada gambar 3.25, lebih mengoptimalkan ruang dengan menampilkan *date picker* dalam format yang lebih kompak menggunakan *input field* standar, dan daftar transaksi ditampilkan dalam format *list* dan *card-design* yang lebih *streamlined* dengan informasi yang terorganisir secara vertikal.

## E Halaman Transaksi hari ini




Gambar 3.26. Halaman Transaksi hari ini versi lama




Gambar 3.27. Halaman Transaksi hari ini versi terbaru

Pada lampiran 3.26, terlihat tampilan desain versi lama masih cukup ok dan bagus. Namun sedikit kekurangan yaitu tidak ada informasi tanggal, dan header dari halaman itu sendiri. Pada desain yang baru pada lampiran 3.27, penulis hanya merubah sedikit beberapa bagian serta warna dari halaman tersebut. Pada desain baru, penulis menambahkan tanggal di *header*, indikator ”Hari Ini” pada *card* total transaksi, dan mata uang ”IDR” pada *grand total*. Sehingga hal ini menambah nilai fungsi dari halaman tersebut dan meningkatkan *user experience*

### 3.3.5 Pengerjaan Sistem Notifikasi Real-time Pesanan Online

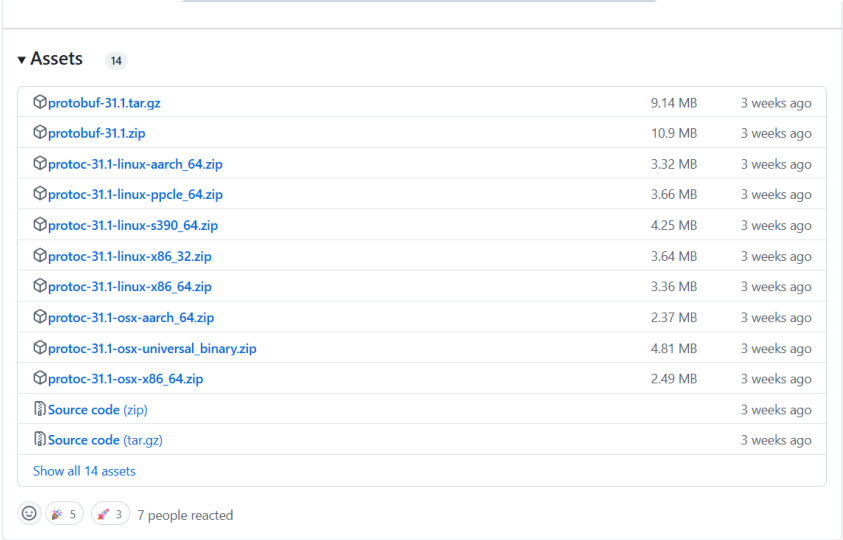
Dalam pengembangannya, penulis perlu mempelajari terlebih dahulu terkait teknologi gRPC (Google Remote Procedure Call) dan socket server. Ini adalah pengalaman pertama penulis dalam mengerjakan komunikasi socket server dengan backend menggunakan protokol gRPC. Supervisor meminta untuk menggunakan gRPC karena keunggulannya dalam hal efisiensi, optimalisasi penggunaan sumber daya, dan kecepatan komunikasi melalui direct service-to-service communication. Menurut dokumentasi resmi gRPC, ”Pada gRPC, aplikasi klien dapat langsung memanggil method pada aplikasi server di mesin yang berbeda seolah-olah seperti objek lokal, sehingga memudahkan pembuatan aplikasi dan layanan terdistribusi” [7]. Dalam FAQ resmi gRPC juga dijelaskan bahwa ”gRPC adalah framework remote procedure call (RPC) modern dan open source yang dapat berjalan di mana saja. gRPC memungkinkan aplikasi klien dan server berkomunikasi secara transparan, dan memudahkan pembuatan sistem yang terhubung”[8]. Dengan ini dapat disimpulkan bahwa gRPC adalah framework RPC modern yang memungkinkan komunikasi transparan antar aplikasi terdistribusi dengan performa tinggi dan efisiensi yang optimal.

Sedangkan socket server merupakan teknologi fundamental dalam komunikasi jaringan. Dalam dokumentasi oracle java [9], socket server adalah socket yang diimplementasikan melalui kelas ServerSocket, yang memungkinkan server untuk mendengarkan dan menerima koneksi dari klien. Socket server berperan sebagai endpoint komunikasi yang menunggu permintaan koneksi dari klien dan kemudian menangani komunikasi dua arah setelah koneksi berhasil dibuat. Menurut Stevens et al., [10] server socket dapat melayani beberapa klien secara bersamaan dengan menciptakan dedicated socket yang unik untuk setiap koneksi klien dalam proses thread pemrosesan terpisah untuk setiap klien. Dengan demikian, socket server bukan hanya sebagai endpoint pasif yang menunggu koneksi, tetapi juga sebagai pengatur untuk mengelola multiple session komunikasi secara bersamaan dengan mempertahankan integritas dan isolasi setiap koneksi klien.

#### A Menginstall Dependencies dan tools untuk gRPC

Berdasarkan dokumentasi resmi gRPC, penulis perlu menginstall beberapa tools dan dependensi yang diperlukan untuk mengimplementasikan gRPC. Yang

pertama adalah menginstall Protocol Buffer Compiler (protoc) yang berfungsi untuk mengkompilasi file .proto menjadi kode yang dapat digunakan dalam bahasa pemrograman Go. Ini diinstall melalui github protocolbuffers/protobuf/releases.




Gambar 3.28. github protoc

Gambar 3.28 menunjukkan halaman *releases* dari repositori GitHub `protocolbuffers/protobuf` yang menyediakan berbagai versi *Protocol Buffer Compiler* (protoc) untuk sistem operasi yang berbeda. Pada gambar terlihat daftar *assets* dari versi 31.1 dengan berbagai format file seperti `.tar.gz` dan `.zip` untuk arsitektur Linux, macOS, dan Windows. Setelah diunduh lalu di ekstrak dan set *environment path* ke folder bin yang ada di file tersebut, sehingga protoc dapat di jalankan. Selanjutnya penginstalan beberapa 2 dependensi yang diperlukan yaitu protoc-gen-go yang berfungsi untuk menghasilkan struktur data Go dari definisi *message* dalam file .proto dan protoc-gen-go-grpc yang berfungsi untuk menghasilkan kode client dan server gRPC.

```
PS C:\Code\Mitra-Integrasi-Digital\Backend Moni\golang-api> go install google.golang.org/protobuf/cmd/protoc-gen-go@latest
```

Gambar 3.29. Install protoc-gen-go

Lampiran 3.29 merupakan gambar yang berisi potongan perintah untuk melakukan install *protoc-gen-go*.

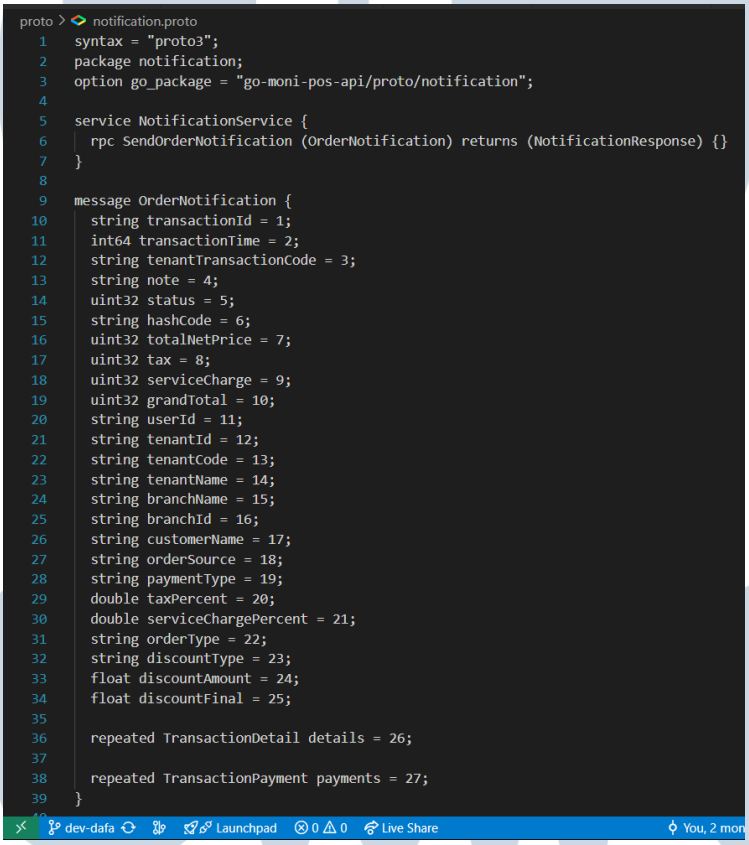
```
PS C:\Code\Mitra-Integrasi-Digital\Backend Moni\golang-api> go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest
```

Gambar 3.30. Install protoc-gen-go-grpc

Sedangkan pada lampiran 3.30 merupakan gambar yang berisi potongan perintah untuk melakukan install *protoc-gen-go-grpc*.

## B Implementasi gRPC di Golang

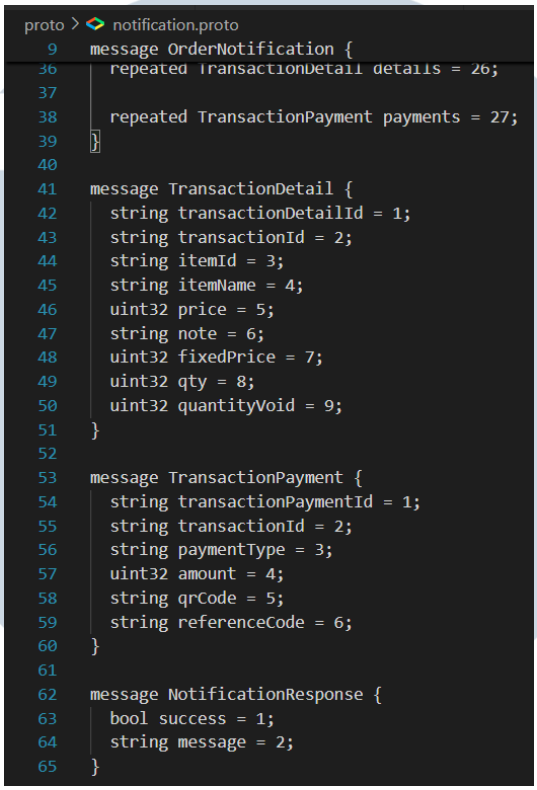
Setelah semua tools dan dependensi yang di butuhkan telah terinstall, lanjut ke tahap pengembangan. Disini penulis membuat struktur data dari file notifikasi proto terlebih dahulu. Berikut struktur datanya




Gambar 3.31. Struktur data notifikasi proto.1.1

Gambar 3.31 menampilkan definisi *service* gRPC dan *message* utama untuk notifikasi. `NotificationService` mendefinisikan *RPC method* `SendOrderNotification` yang menerima parameter `OrderNotification` dan mengembalikan `NotificationResponse`. `OrderNotification message` berisi informasi komprehensif tentang transaksi meliputi ID transaksi, waktu transaksi, kode transaksi tenant, status, informasi finansial (harga bersih, pajak, biaya layanan, total), informasi tenant dan cabang, detail customer, jenis pembayaran, persentase pajak dan biaya layanan, tipe pesanan, serta informasi

diskon. Struktur ini memungkinkan transfer data yang lengkap dan terstruktur untuk sistem notifikasi *real-time*.




Gambar 3.32. Struktur data notifikasi proto.1.2

Pada gambar 3.32 menunjukkan definisi *message* untuk komponen detail notifikasi pesanan online. `OrderNotification` berfungsi sebagai *root message* yang berisi array dari `TransactionDetail` dan `TransactionPayment` untuk menyimpan informasi lengkap pesanan. `TransactionDetail` mendefinisikan struktur data untuk setiap item dalam pesanan meliputi ID detail, ID transaksi, informasi produk (ID, nama, harga), catatan khusus, harga tetap, kuantitas, dan jumlah yang dibatalkan. Sedangkan `TransactionPayment` menyimpan informasi pembayaran seperti ID pembayaran, jenis pembayaran, jumlah, kode QR, dan kode referensi.

```
PS C:\Code\Mitra-Integrasi-Digital\Backend Moni\golang-api> protoc --go_out=. --go_opt=paths=source_relative \
>>     --go-grpc_out=. --go-grpc_opt=paths=source_relative \
>>     proto/notification/notification.proto
```

Gambar 3.33. Build file Proto

Setelah file Protocol Buffer (.proto) telah dibuat, lalu kemudian file tersebut

dijalankan untuk menghasilkan 2 file go yaitu notification.pb.go yang berisi struct-struct Go untuk semua message, dan notification-grpc.pb.go yang berisi interface dan implementasi gRPC. Untuk perintahnya dalam dilihat pada lampiran 3.33

Kemudian penulis mengimplementasikan gRPC client di backend Go untuk mengirim notifikasi pesanan ke socket server. Implementasi ini dibuat dalam bentuk service yang terpisah untuk memudahkan maintenance dan testing. NotificationService dibuat sebagai service khusus yang menangani komunikasi gRPC dengan socket server. Untuk prosesnya yang pertama ada

1. Connection Management. Sistem membuat koneksi ke socket server menggunakan grpc.Dial(). Setiap request dilengkapi dengan context timeout 5 detik untuk mencegah hanging connection.

2. Data Transformation. sebelum dikirim, data transaction perlu di *convert* dari model internal Go ke format Protocol Buffer.

3. Pengiriman. Setelah data siap, sistem mengirim `OrderNotification` request ke socket server melalui method `SendOrderNotification()`. Response dari socket server kemudian dihandle dengan logging untuk memeriksa apakah terkirim atau tidak.

```go
// SendOrderNotification to socket server
func (s *NotificationService) SendOrderNotification(transaction models.Transaction) error {

    conn, err := grpc.Dial(s.socketServerAddr, grpc.WithTransportCredentials(insecure.NewCredentials()))
    if err != nil {
        utils.Logger.Sugar().Errorf("Failed to connect to socket server: %v", err)
        return err
    }
    defer conn.Close()

    client := notification.NewNotificationServiceClient(conn)
    ctx, cancel := context.WithTimeout(context.Background(), time.Second*5)
    defer cancel()

    var transactionDetails []*notification.TransactionDetail
    for _, detail := range transaction.TransactionDetail {
        transactionDetails = append(transactionDetails, &notification.TransactionDetail{
            TransactionDetailId: detail.TransactionDetailId,
            TransactionId:       detail.TransactionId,
            ItemId:              detail.ItemId,
            ItemName:            detail.ItemName,
            Price:               uint32(detail.Price),
            Note:                detail.Note,
            FixedPrice:          uint32(detail.FixedPrice),
            Qty:                 uint32(detail.Qty),
            QuantityVoid:        uint32(detail.QuantityVoid),
        })
    }

    // Convert to gRPC compatible format
    var transactionPayments []*notification.TransactionPayment
    for _, payment := range transaction.TransactionPayment {
        transactionPayments = append(transactionPayments, &notification.TransactionPayment{
            TransactionPaymentId: payment.TransactionPaymentId,
            TransactionId:        payment.TransactionId,
            PaymentType:          payment.PaymentType,
            Amount:               uint32(payment.Amount),
```

Gambar 3.34. Potongan kode dari notifikasi servis

Pada gambar 3.34 dapat dilihat implementasi *method* `SendOrderNotification` yang menangani proses pengiriman notifikasi. *Method* ini dimulai dengan membuat koneksi gRPC ke *socket server* menggunakan `grpc.Dial()` dengan konfigurasi *insecure credentials*. Selanjutnya dilakukan transformasi data dari model `Transaction` internal menjadi format *Protocol Buffer* yang kompatibel dengan gRPC. Proses transformasi meliputi konversi `TransactionDetail` dan `TransactionPayment` ke dalam struktur `notification.TransactionDetail` dan `notification.`

Kemudian *notification service* tersebut akan dipanggil pada saat *create-transaction* jika *type-order* sama dengan `SELF_ORDER`, maka data akan diteruskan ke *socket server* dan *socket server* akan mengirimkannya ke *client* yang terhubung.

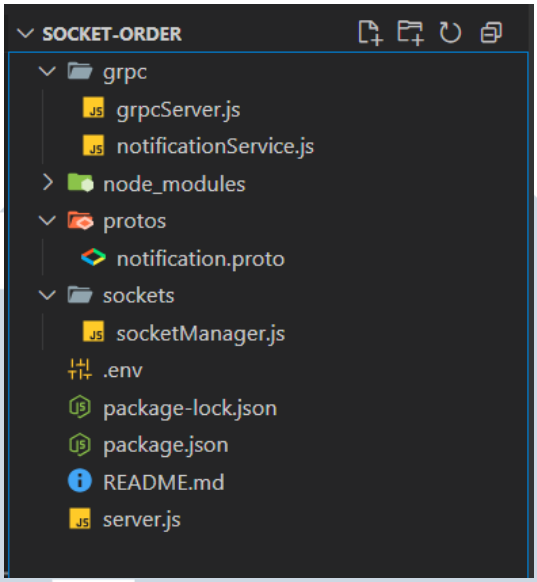
```
// If order source is SELF_ORDER
if strings.ToUpper(payload.Info.OrderSource) == "SELF_ORDER" {
    /// Send notification to the socket server
    if notificationService != nil {
        err := notificationService.SendOrderNotification(trx)
        if err != nil {
            utils.Logger.Sugar().Errorf("Failed to send notification to socket server: %v", err)
            /// Proceed even if the notification fails
        } else {
            utils.Logger.Sugar().Infof("Notification sent to socket server for transaction %s", trx.TransactionId)
        }
    } else {
        utils.Logger.Sugar().Warnf("Notification service not initialized, unable to send notification for transaction %s", trx.TransactionId)
    }
}
fmt.Print("trx: ", trx)
c.JSON(http.StatusOK, trx)
```

Gambar 3.35. Kode jika tipe order adalah self order

Gambar 3.35 menunjukkan implementasi *trigger* notifikasi dalam *endpoint create transaction*. Sistem melakukan pengecekan kondisi `payload.Info.OrderSource == "SELF_ORDER"` untuk memastikan notifikasi hanya dikirim untuk pesanan yang berasal dari *self-order*. Jika kondisi terpenuhi dan `notificationService` telah terinisialisasi, sistem akan memanggil `SendOrderNotification()` dengan data transaksi yang baru dibuat. Implementasi ini dilengkapi dengan *error handling* yang mencatat log jika pengiriman gagal namun tetap melanjutkan proses transaksi, serta pesan sukses ketika notifikasi berhasil dikirim ke *socket server*.

## C Socket server

Socket server dibuat sebagai penghubung antara backend gRPC server (Go) dengan Moni pos client. Socket ini dibuat dengan Node.js dan menerapkan gRPC.




Gambar 3.36. Struktur file Socket Server

Gambar 3.36 menampilkan struktur organisasi file dalam proyek *socket server* yang mengimplementasikan arsitektur modular. Struktur ini terdiri dari folder `grpc` untuk komponen gRPC, folder `protos` yang menyimpan definisi *Protocol Buffer*, folder `sockets` untuk manajemen *WebSocket*, serta file konfigurasi seperti `package.json` dan `server.js` sebagai *entry point*. Untuk lebih lengkapnya pada berikut:

1. Main Server (app.js) Merupakan file utama yang menginisialisasi HTTP server, Socket.IO, dan gRPC server secara bersamaan. Server berjalan di dua port 3000 untuk HTTP/WebSocket dan port 50051 untuk gRPC communication.

2. Socket Manager (socketManager.js) Berfungsi untuk mengelola koneksi WebSocket dari client dan mengorganisir client berdasarkan tenant ID menggunakan sistem room. Setiap tenant memiliki room terpisah agar notifikasi hanya diterima oleh merchant yang tepat.

3. gRPC Service Handler (notificationService.js) Berfungsi sebagai handler untuk menerima request dari backend Go melalui gRPC protocol. Ketika menerima notifikasi pesanan, handler ini akan mem-broadcast pesan ke room tenant yang sesuai melalui Socket.IO.

4. gRPC Server Configuration (grpcServer.js) Mengkonfigurasi gRPC server

dengan memuat file Protocol Buffer dan mendefinisikan service yang tersedia. File ini juga menangani binding server ke port yang ditentukan.

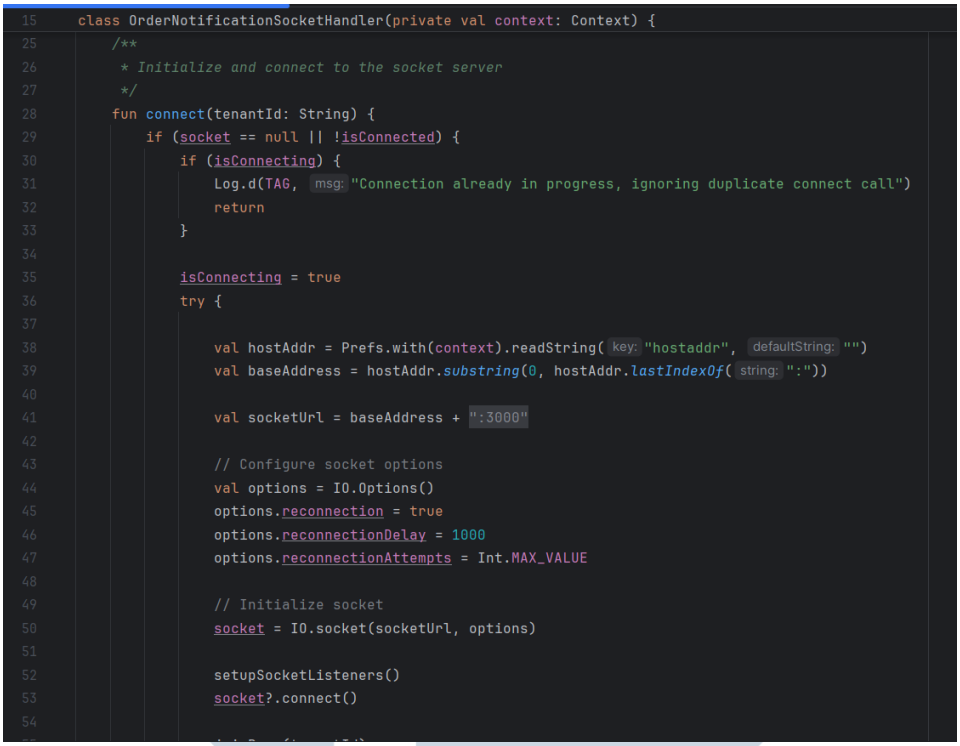
### D Pada sisi android Moni POS

Setelah *socket server* berhasil diimplementasikan, langkah selanjutnya adalah mengintegrasikan sistem notifikasi *real-time* ke dalam aplikasi Android MoniPos. Integrasi ini memungkinkan aplikasi untuk menerima dan memproses notifikasi pesanan secara *real-time* tanpa perlu melakukan *polling* data secara berkala ke server. Nantinya Android dapat mendapatkan notifikasi terkait orderan yang masuk, dapat memproses status orderan yang masuk dari mulai proses *order*, hingga pesanan selesai. Implementasi ini menggunakan teknologi *Socket.IO client* untuk Android yang memungkinkan komunikasi dua arah antara aplikasi dan *socket server*. Pada Android terdapat 3 komponen file penting dalam pengembangan fitur notifikasi *order online*, yaitu *OrderNotificationSocketHandler*, *Order Online Activity*, dan *Detail Order Online Activity*.

#### D.1 OrderNotificationSocketHandler

File ini adalah komponen utama yang akan menangani koneksi WebSocket antara aplikasi Android dengan socket server. Tujuannya untuk untuk mengelola koneksi ke socket server, menerima data dari server, menangani error dan reconnection otomatis, mengatur status koneksi (connected, disconnected, connecting), memproses response dari server socket, serta mengelola timeout. Untuk lebih lanjut, file ini bertanggung jawab untuk:

1. Menginisialisasi koneksi Socket.IO dengan konfigurasi auto-reconnection
2. Bergabung ke room tenant sesuai dengan ID merchant
3. Mendengarkan event `new_order` dari socket server
4. Memproses data pesanan yang diterima dan menampilkan notifikasi push



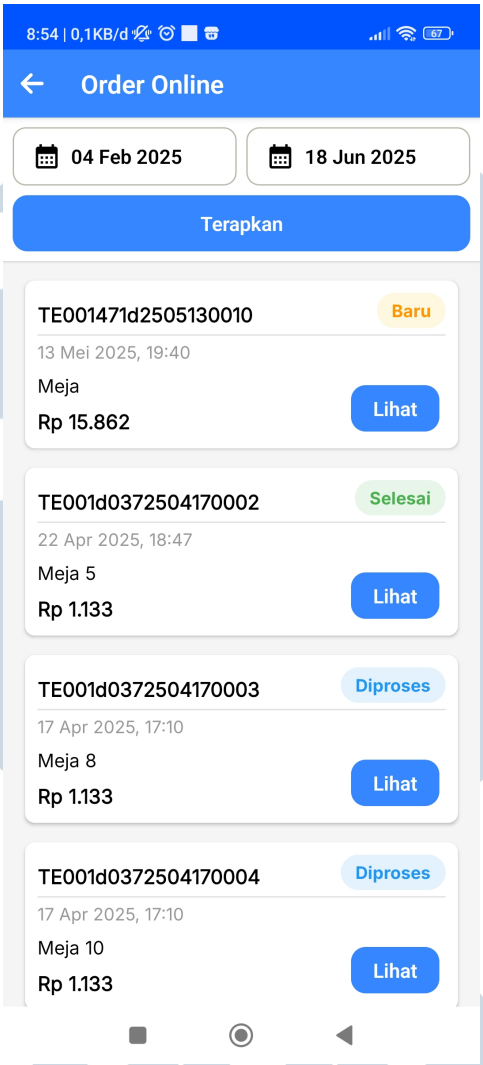
```
class OrderNotificationSocketHandler(private val context: Context) {
    /**
     * Initialize and connect to the socket server
     */
    fun connect(tenantId: String) {
        if (socket == null || !isConnected) {
            if (isConnecting) {
                Log.d(TAG, msg: "Connection already in progress, ignoring duplicate connect call")
                return
            }

            isConnecting = true
            try {

                val hostAddr = Prefs.with(context).readString( key: "hostaddr",  defaultString: "")
                val baseAddress = hostAddr.substring(0, hostAddr.lastIndexOf( string: ":"))

                val socketUrl = baseAddress + ":3000"

                // Configure socket options
                val options = IO.Options()
                options.reconnection = true
                options.reconnectionDelay = 1000
                options.reconnectionAttempts = Int.MAX_VALUE

                // Initialize socket
                socket = IO.socket(socketUrl, options)

                setupSocketListeners()
                socket?.connect()
```

Gambar 3.37. Potongan Kode OrdernotificationSocketHandler

Gambar 3.37 menunjukkan implementasi *method* `connect()` dalam `OrderNotificationSocketHandler` yang menangani proses koneksi ke *socket server*. *Method* ini dimulai dengan pengecekan status koneksi untuk mencegah koneksi duplikat, kemudian mengambil alamat *host* dari `SharedPreferences` dan membangun URL *socket* dengan port 3000. Konfigurasi *Socket.IO* menggunakan `IO.Options()` dengan pengaturan *auto-reconnection*, *reconnection delay* 1000ms, dan *reconnection attempts* maksimal. Setelah *socket* diinisialisasi, sistem memanggil `setupSocketListeners()` untuk mengatur *event listeners* dan `socket.connect()` untuk memulai koneksi ke server.

## D.2 Order Online Activity

Activity ini dibuat untuk menampilkan list-list orderan yang masuk. Data diambil dari API, dengan status 11 untuk order online yang baru masuk, 12 untuk order online yang sedang di proses, 8 untuk orderan yang sudah selesai. Selain itu, dihalaman ini juga terdapat filter untuk memfilter data orderan berdasarkan tanggal. Untuk tampilannya seperti berikut:



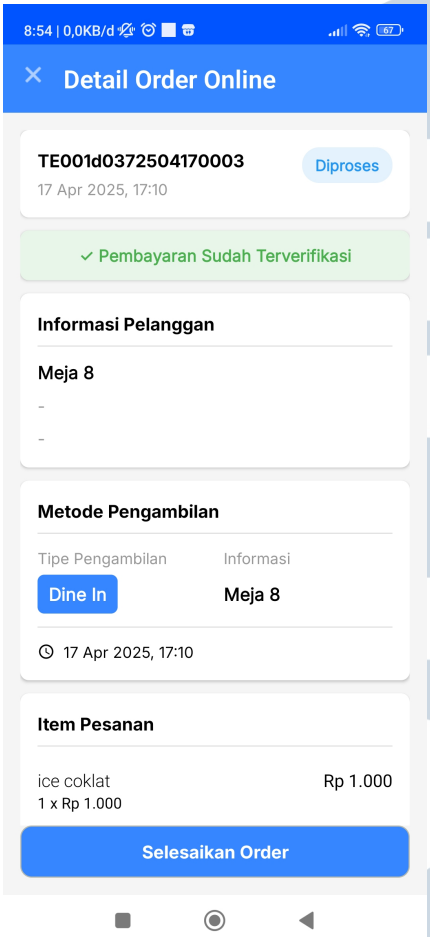

Gambar 3.38. Halaman Order Online

Gambar 3.38 menunjukkan implementasi halaman *Order Online* yang menampilkan daftar pesanan *online* dengan berbagai status. Halaman ini dilengkapi dengan *date range picker* di bagian atas untuk memfilter pesanan berdasarkan periode waktu tertentu. Setiap item pesanan ditampilkan dalam format *card* yang berisi informasi lengkap seperti nomor transaksi, tanggal dan waktu pesanan, nama meja, total pembayaran, serta tombol ”Lihat” untuk melihat detail pesanan.

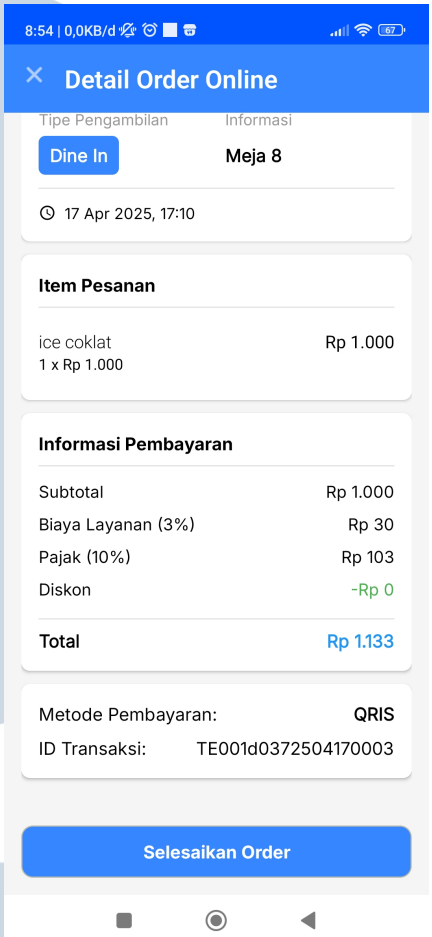
## D.3 Detail Order Online Activity

Activity ini dibuat untuk menampilkan detail dari pesanan Self order yang masuk. Tak hanya itu, halaman ini juga dapat melakukan update status terkait orderan. Untuk statusnya terdapat 3 jenis yaitu 11 untuk orderan yang baru masuk,

12 untuk pesanan yang sedang di proses, dan 8 untuk pesanan yang sudah selesai dibuat. Untuk halamannya seperti pada gambar berikut:




Gambar 3.39. Halaman Detail Order online 1.1




Gambar 3.40. Halaman Detail Order Online 1.2

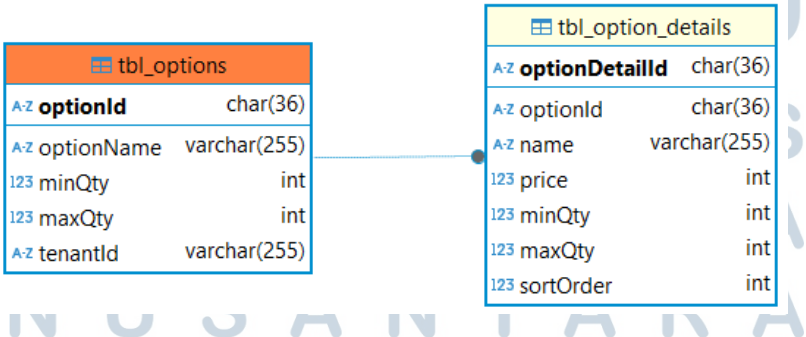
Gambar 3.39 dan 3.40 merupakan gambar dari halaman detail pesanan *online* yang menampilkan informasi lengkap tentang pesanan customer. Halaman ini terdapat beberapa section: *Informasi Pelanggan* yang menampilkan nomor meja, *Metode Pengambilan* yang menunjukkan tipe pesanan beserta informasi meja dan waktu pesanan, serta *Item Pesanan* yang berisi daftar produk yang dipesan. Di bagian bawah terdapat *Informasi Pembayaran* yang mencakup breakdown biaya serta informasi metode pembayaran dan ID transaksi, dan terdapat Tombol ”Selesaikan Order” untuk mengubah status pesanan sesuai dengan progress.

### 3.3.6 Pengerjaan item-option

Supervisor menginginkan pada setiap menu tidak hanya berisi informasi dasar menu saja, tapi juga ingin dilengkapi dengan option, seperti topping, tingkat level pedas. Setiap option juga memiliki quantity minimal, maximal, juga harga tersendiri yang dapat ditambahkan ke harga dasar menu. Fitur ini nantinya akan diselaraskan dengan website self-order yang sedang dikembangkan oleh supervisor penulis. Dalam pengerjaannya penulis perlu menambahkan tabel baru pada database yang sudah ada, yaitu tabel option, dan tabel detail option. Penulis juga perlu memodifikasi beberapa tabel yang sudah ada. Setelah konfigurasi option telah selesai, penulis juga perlu menampilkan option tersebut di semua jenis struk, baik reprint, transaksi, dan checker untuk dapur.

#### A Tabel yang diperlukan

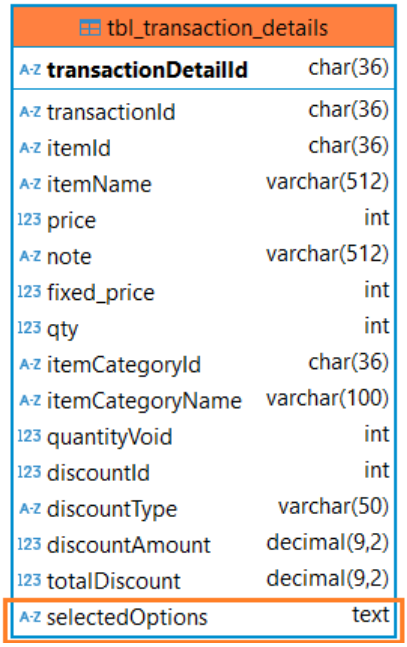
Dalam pengembangan fitur ini, diperlukan 2 tabel baru seperti yang sudah disebutkan sebelumnya, yaitu tabel option dan tabel detail-option. Tabel option berfungsi sebagai *parent* tabel yang akan menyimpan informasi dasar dari setiap grup option. Tabel ini memiliki kolom-kolom utama seperti nama option (misalnya ”Topping”, ”Level Pedas”, ”Ukuran”), minimal dan maksimal quantity yang menentukan batasan pilihan customer. Sebagai contoh, option topping memiliki minimal 0 (opsional) dan maksimal 3 pilihan, sedangkan option level pedas memiliki minimal 1 dan maksimal 1 (wajib dipilih). Kemudian juga tabel ini memiliki relasi ke tabel detail-option melalui foreign key yang menghubungkan setiap option dengan detail pilihan spesifiknya.



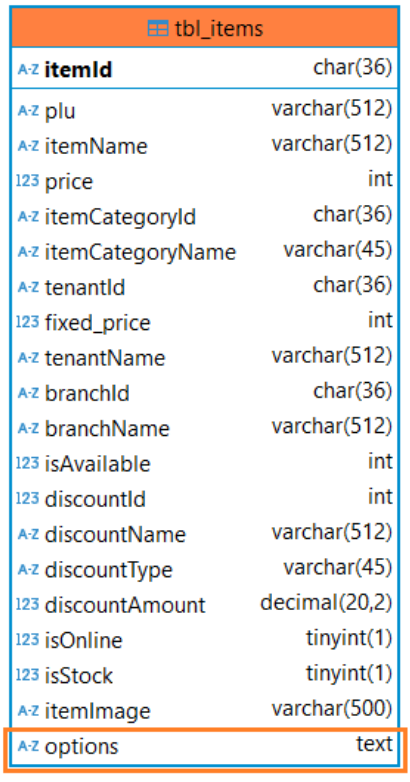

Gambar 3.41. Tabel option dan Detail-option

Kemudian tabel detail-option berfungsi sebagai tabel detail yang menyimpan pilihan-pilihan spesifik untuk setiap option. tabel ini memiliki kolom

id-detail-option sebagai primary key, kolom optionId sebagai foreign-key untuk menuju tabel option, kemudian terdapat kolom nama, harga, minimal dan maximal quantity untuk menentukan banyaknya satuan item option yang bisa dipilih, misal user memilih item cheese 2x maka bisa dibilang seperti ekstra double keju. Kemudian juga ada kolom sort order yang digunakan untuk menyimpan urutan index option agar dapat ditampilkan sesuai dengan prioritas yang diinginkan merchant. Selanjut untuk modifikasi tabel, penulis perlu mengubah 2 tabel yang sudah ada yaitu tabel detail transaksi dan tabel item. Pada tabel detail transaksi penulis menambahkan kolom *SelectedOption* untuk memasukan data opsi yang dipilih oleh user. Kemudian untuk tabel item, penulis menambahkan kolom options yang akan menyimpan id-option, digunakan untuk mengetahui option apa saja yang dimiliki oleh item tersebut.




Gambar 3.42. Table transaction detail




Gambar 3.43. Tabel items

## B Konfigurasi option pada Golang

Setelah struktur database yang diperlukan untuk fitur option telah dibuat, selanjutnya adalah mengkonfigurasikan option pada sisi backend golang.

Implementasi ini meliputi membuat dan modifikasi model data yang sudah ada, lalu menambah logic terkait konfigurasi option.

## B.1 Struktur Model

Untuk models penulis melakukan hal yang sama seperti pada tabel, yaitu menambah model option dan detail option, memodifikasi model items dan model detail transaction. Model dibuat, merepresentasikan tabel yang ada dalam database. Berikut untuk model dari option dan option-detail.


Gambar 3.44. Model Option

Gambar 3.44 menunjukkan struktur model `Option` yang merepresentasikan tabel *options* dalam database. Model ini menggunakan *struct tags* untuk konfigurasi JSON dan GORM ORM, dengan field `OptionId` sebagai *primary key*, `OptionName` untuk nama opsi, `MinQty` dan `MaxQty` untuk menentukan batas minimum dan maksimum kuantitas yang dapat dipilih, serta `TenantId` untuk mengasosiasikan opsi dengan tenant tertentu. Model ini juga memiliki relasi `OptionDetails` yang menggunakan *foreign key* `OptionId` untuk menghubungkan dengan detail opsi.

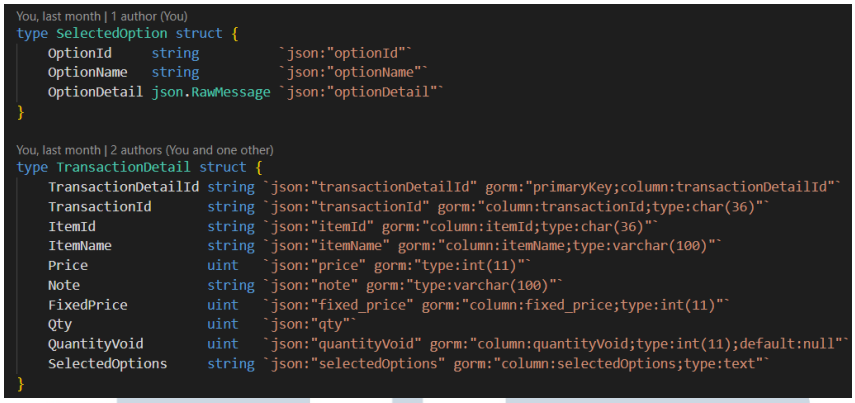
```
You, last week | 1 author (You)
package models

import (
    "github.com/segmentio/ksuid"
    "gorm.io/gorm"
)

You, last week | 1 author (You)
type OptionDetail struct {
    OptionDetailId string `json:"optionDetailId" gorm:"primaryKey;column:optionDetailId"`
    OptionId       string `json:"optionId" gorm:"column:optionId;type:char(36)"`
    Name           string `json:"name" gorm:"column:name;type:varchar(255)"`
    Price          int    `json:"price" gorm:"column:price;default:0"`
    MinQty         int    `json:"minQty" gorm:"column:minQty;default:0"`
    MaxQty         int    `json:"maxQty" gorm:"column:maxQty;default:0"`
    SortOrder      int    `json:"sortOrder" gorm:"column:sortOrder;default:0"`
}

func (OptionDetail) TableName() string {
    return "tbl_option_details"
}

func (detail *OptionDetail) BeforeCreate(tx *gorm.DB) (err error) {
    detail.OptionDetailId = ksuid.New().String()
    return
}
```

Gambar 3.45. Model OptionDetail

Gambar 3.45 menampilkan struktur model `OptionDetail` yang merepresentasikan detail dari setiap opsi. Model ini berisi field `OptionDetailId` sebagai *primary key*, `OptionId` sebagai *foreign key* yang menghubungkan dengan tabel *options*, `Name` untuk nama detail opsi, `Price` untuk harga tambahan jika ada, `MinQty` dan `MaxQty` untuk batasan kuantitas, serta `SortOrder` untuk mengatur urutan tampilan.

Penulis menggunakan GORM sebagai ORM (*Object-Relational Mapping*) untuk mempermudah interaksi dengan database. Setiap model dilengkapi dengan *tag* GORM yang mendefinisikan struktur kolom database, tipe data, dan *constraint* yang diperlukan. Field `OptionDetails` dalam *struct* `Option` menggunakan relasi `foreignKey:optionId` untuk menghubungkan dengan tabel *detail-option*, memungkinkan pemuatan data relasi secara otomatis menggunakan fitur *preload* GORM.

Kemudian juga terdapat fungsi `BeforeCreate` yang akan secara otomatis meng-*generate* ID menggunakan KSUID (*K-Sortable Unique Identifier*). Hal ini bertujuan agar setiap *record* memiliki ID yang unik. Kemudian untuk model *transaction detail* penulis menambahkan `selectedoption` bertipe *string* yang digunakan untuk menyimpan data JSON. Berikut model dari *transaction detail*:




Gambar 3.46. model transactionDetail

Gambar 3.46 menunjukkan modifikasi pada model `TransactionDetail` dengan penambahan field `SelectedOptions` bertipe *string* yang digunakan untuk menyimpan data JSON pilihan opsi yang dipilih customer. Field ini menggunakan *GORM tag* `column:selectedOptions;type:text` untuk mendefinisikan kolom database dengan tipe *text* yang dapat menampung data JSON berukuran besar.

Lalu untuk model *items* penulis menambahkan `options` dengan tipe data *string* untuk menyimpan data berupa JSON *array*. Berikut lampiran modelnya:


Gambar 3.47. Model items

Gambar 3.47 memperlihatkan penambahan field `Options` pada model `Item` dengan tipe data *string* dan *GORM tag* `column:options;type:text`. Field ini dirancang untuk menyimpan data JSON yang berisi informasi opsi-opsi yang tersedia untuk setiap item menu.

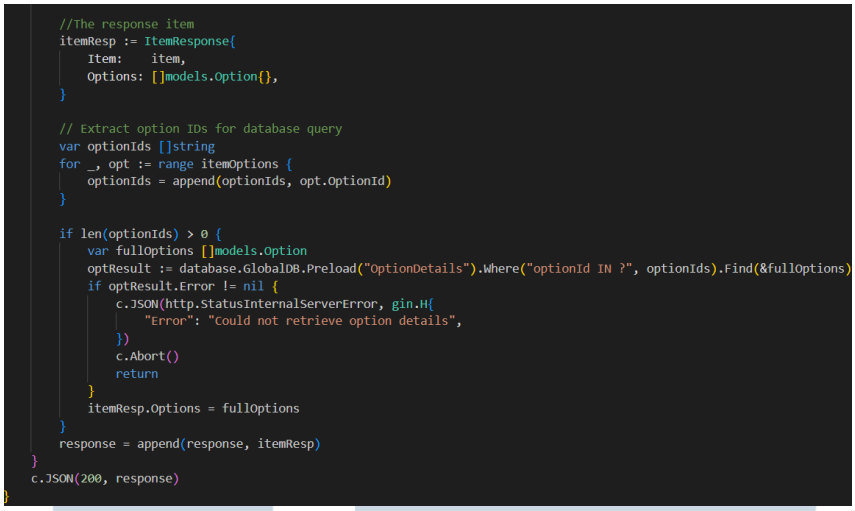
## C Penambahan Logic terkait fitur option

Setelah memodifikasi models, selanjut penulis membuat logic untuk memasukan fitur option kedalam menu. Salah satu implementasinya adalah *API GetTenantItem* yang dimodifikasi untuk memuat *option* data secara lengkap beserta *detail-option*. *Api* ini digunakan untuk mengambil menu berdasarkan tenant. Dalam implementasinya, penulis mencoba menerapkan preloading untuk menghindari permasalahan *N+1 query problem* yang dapat menurunkan performa.

```go
func GetTenantItem(c *gin.Context) {
    var items []models.Item
    tenantId := c.Param("tenantId")
    result := database.GlobalDB.Where("tenantId = ?", tenantId).Preload("ItemCategory").Find(&items)
    if result.Error == gorm.ErrRecordNotFound {
        c.JSON(http.StatusNotFound, gin.H{
            "Error": "Item Not Found",
        })
        c.Abort()
        return
    }
    if result.Error != nil {
        c.JSON(http.StatusInternalServerError, gin.H{
            "Error": "Could Not Get Item",
        })
        c.Abort()
        return
    }

    type ItemResponse struct {
        models.Item
        Options []models.Option `json:"options"`
    }
    var response []ItemResponse

    for _, item := range items {
        itemOptions, err := item.GetItemOptions()
        if err != nil {      You, last month • Refactor GetTenantItem to return item options f…
            c.JSON(http.StatusInternalServerError, gin.H{
                "Error": "Could not parse item options",
            })
            c.Abort()
            return
        }
```

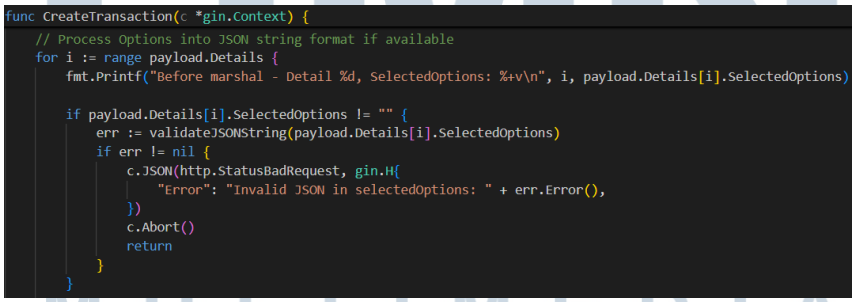
Gambar 3.48. Potongan kode get-tenantItem-1.1

Gambar 3.48 menunjukkan bagian awal dari fungsi `GetTenantItem` yang melakukan *query* database untuk mengambil item berdasarkan `tenantId` dengan menggunakan *preload* `ItemCategory` untuk memuat data kategori secara bersamaan. Kode ini juga mendefinisikan *struct* `ItemResponse` yang menggabungkan data `models.Item` dengan field `Options` bertipe `[]models.Option` untuk menampung data opsi. Dalam *loop* pemrosesan setiap item, sistem memanggil *method* `GetItemOptions()` untuk mengambil opsi yang terkait dengan item tersebut, dengan *error handling* yang akan mengembalikan respons error jika terjadi kegagalan dalam *parsing* opsi item.




Gambar 3.49. Potongan kode get-tenantItem-1.2

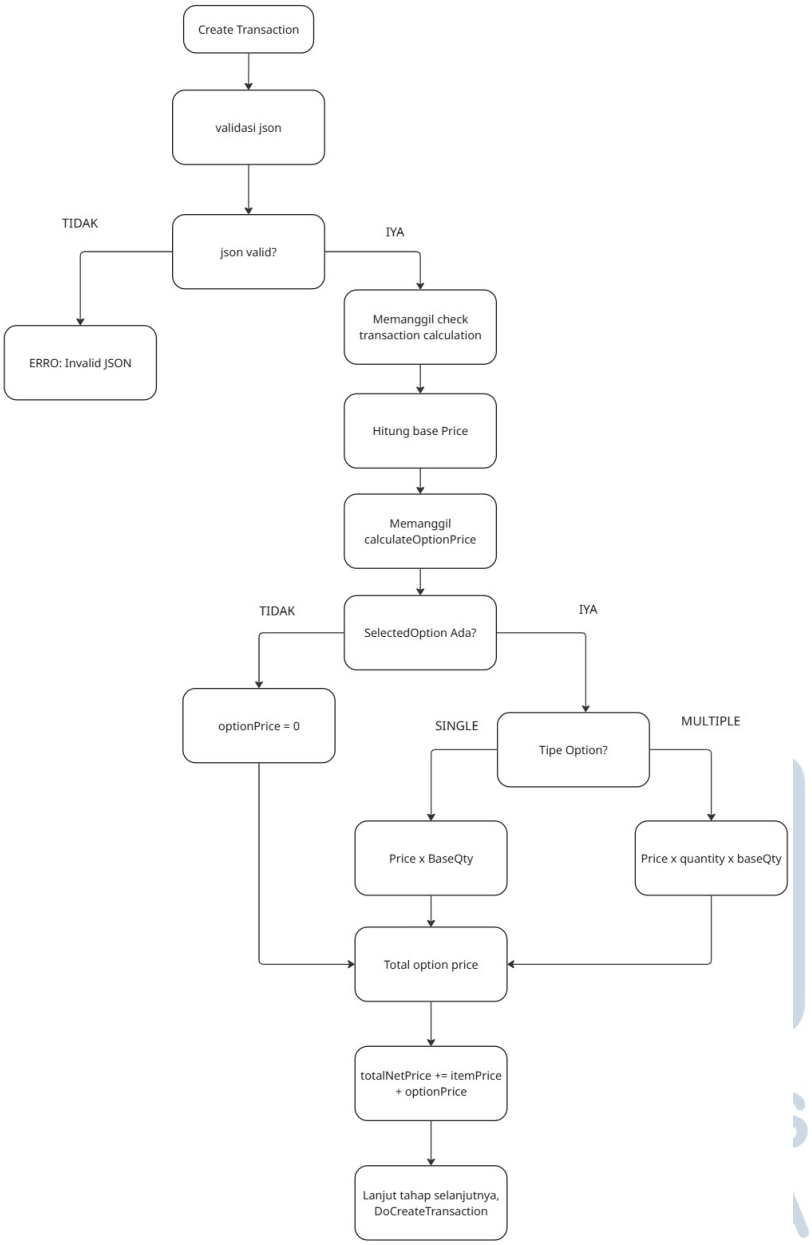
Gambar 3.49 memperlihatkan kelanjutan proses dimana sistem mengekstrak `optionIds` dari setiap item dan melakukan *query* database untuk mengambil data opsi lengkap beserta *detail-option* menggunakan *preload* `OptionDetails`. Implementasi ini menggunakan kondisi `WHERE optionId IN ?` dengan parameter `optionIds` untuk mengambil semua opsi yang diperlukan dalam satu *query*, sehingga menghindari *N+1 query problem*. Setelah data opsi berhasil diambil, sistem memasukan data tersebut ke `itemResp.Options` dan menambahkannya ke *response array*, kemudian mengembalikan hasil dalam format JSON dengan status `200 OK`. Kemudian implementasi option ke create-transaction. Berikut potongan kodenya:




Gambar 3.50. Potongan kode konfigurasi option pada create-transaction

Gambar 3.50 menunjukkan implementasi validasi JSON *option* dalam fungsi *create transaction*. Pada kode tersebut, terdapat proses validasi dimana sistem melakukan *unmarshal* data JSON *selected options* untuk memastikan format data yang dikirim oleh *client* sesuai dengan struktur yang diharapkan. Jika terjadi

*error* dalam proses *parsing*, sistem akan mengembalikan respons *error* dengan status `400 Bad Request` dan pesan ”Invalid selected options format”. Setelah validasi berhasil, sistem melanjutkan ke fungsi `checkTransactionCalculation` untuk memproses perhitungan harga.




Gambar 3.51. Alur Diagram option pada CreateTransaction

Gambar 3.51 menjelaskan alur *workflow* implementasi fitur *option* dalam proses *create transaction*. Proses dimulai dengan validasi JSON yang dikirim oleh *client*, dimana sistem akan mengembalikan *error* ”Invalid JSON” jika

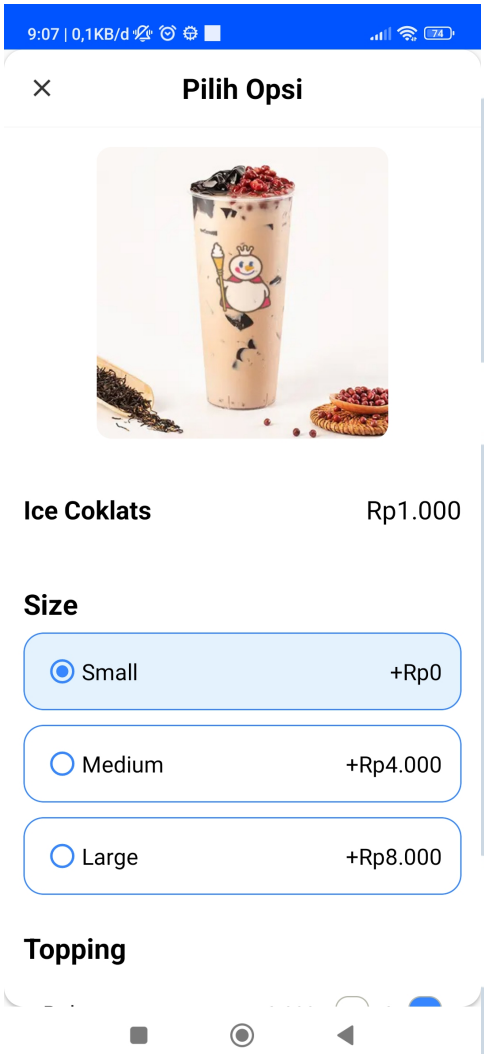
format data tidak sesuai. Setelah validasi berhasil, sistem memanggil fungsi `checkTransactionCalculation` untuk menghitung harga dasar item, kemudian dilanjutkan dengan pemanggilan `calculateOptionPrice` untuk memproses harga tambahan dari opsi yang dipilih. Dalam perhitungan opsi, sistem membedakan antara *single selection* (harga × kuantitas dasar) dan *multiple selection* (harga × kuantitas × kuantitas dasar). Hasil akhir perhitungan adalah `totalNetPrice = itemPrice + optionPrice` yang kemudian diteruskan ke tahap `DoCreateTransaction` untuk menyimpan data transaksi ke database.

### D Pada sisi Android MoniPos

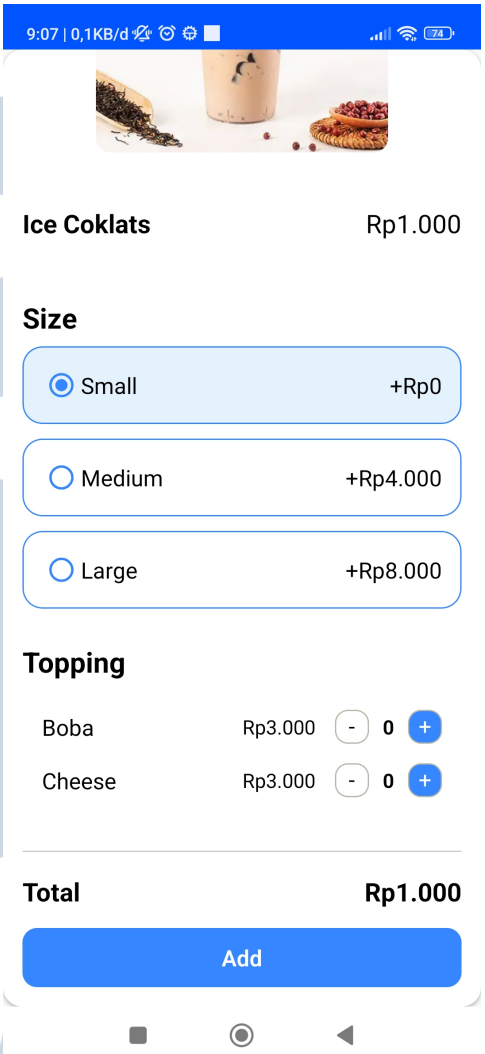
Untuk mengimplementasi option pada android, penulis perlu memodifikasi beberapa file. Yang pertama adalah *ListMenuParser*. Ketika user berhasil login pada monipos, kemudian sistem akan memanggil *API gettenantitem* untuk mendapat data menu yang dimiliki oleh tenant tersebut. Setelah memanggil api tersebut lalu sistem akan memanggil fungsi listmenuparser. Tujuanya adalah untuk mengolah data json sekaligus menyimpan ke *database* lokal android. Disini penulis perlu menambahkan kolom option pada database lokal agar option dapat tersimpan juga di lokal *database*.

Kemudian penulis juga memodifikasi pada file *transactionhelper*. File *TransactionHelper* berfungsi sebagai *generator payload* transaksi yang akan dikirim ke *backend* server. Modifikasi utama yang dilakukan adalah pada method generateTransactionDetails() untuk menangani data selectedOptions yang dipilih customer. Dalam implementasinya, penulis menambahkan logic untuk memproses field selectedOptions yang tersimpan dalam format object menjadi JSON array string sesuai dengan yang diharapkan oleh API backend. Proses konversi ini dilakukan dengan cara mengiterasi setiap option yang dipilih customer, kemudian mengubah struktur data dari format object ke format *array*. Hal ini diperlukan karena data option yang disimpan sementara di cart Android berbentuk object dengan key-value pairs, sedangkan backend *API* mengharapkan format *JSON array string*.

Lalu penulis juga menambahkan halaman *itemOptionDialog*. Halaman ini digunakan untuk memilih opsi ketika sebuah menu memiliki *option*. Berikut untuk tampilan halamannya:




Gambar 3.52. Halaman memilih opsi-1.1




Gambar 3.53. Halaman memilih opsi-1.2

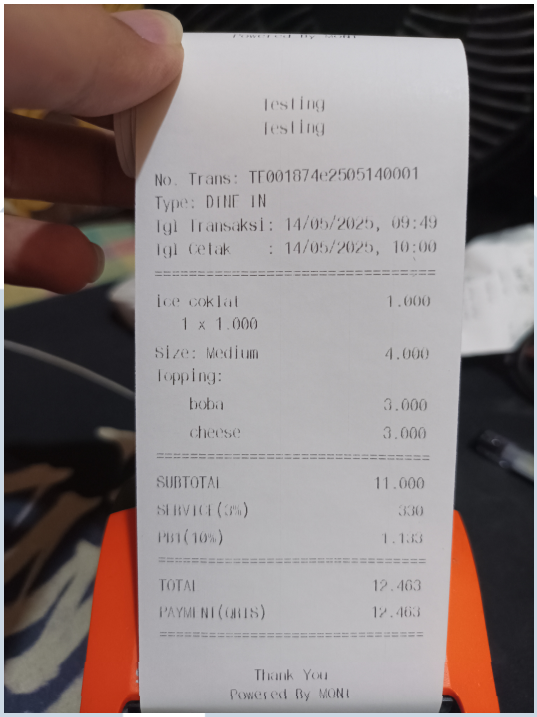
Gambar 3.52 dan 3.53 merupakan tampilan dari halaman item yang memiliki opsi. Terdapat gambar produk, harga dasar item, dan berbagai pilihan opsi yang dapat dikustomisasi oleh pengguna. Untuk kategori ”Size”, sistem menampilkan pilihan dengan *radio button* karena opsi ini memiliki batas minimal dan maksimal 1 (hanya dapat memilih satu ukuran), dengan masing-masing pilihan menunjukkan tambahan harga yang akan dikenakan. Sedangkan untuk kategori ”Topping”, sistem menggunakan *increment* dan *decrement button* (plus dan minus) karena opsi ini dapat dipilih lebih dari satu. Di bagian bawah halaman terdapat total harga yang akan diupdate secara *real-time* setiap kali pengguna mengubah pilihan opsi, serta tombol ”Add” untuk menambahkan item beserta opsi yang dipilih ke dalam keranjang belanja.

Setelah semua selesai penulis juga mengkonfigurasi semua *layout* struk agar menampilkan *option*. Penulis menyesuaikan dan menambahkan seperti data model, pengelolaan data, dan membuat agar opsi tercetak di struk pada ketiga jenis printer tersebut, yaitu *bluetooth*, *sunmi*, dan *verifone*. Proses konfigurasi ini meliputi modifikasi pada *method* `parseSelectedOptions()` di setiap *class printer* untuk memproses data JSON opsi yang terpilih, yang bersumber dari *database*, kemudian mengkonversinya menjadi format teks yang sesuai dengan karakteristik masing-masing printer. Berikut lampiran dari struk dengan opsi:


Gambar 3.54. Struk transaksi pada printer bluetooth dengan *optionItem*


Gambar 3.55. Struk transaksi pada printer verifone dengan *optionItem*

Gambar 3.54 dan 3.55 adalah hasil implementasi fitur *option* pada *output* struk dari printer *Bluetooth* dan *Verifone*. Kedua struk menampilkan informasi opsi yang dipilih customer dengan format yang terstruktur dan mudah dibaca. Disini penulis menambahkan indentasi yang jelas untuk membedakan antara item utama dengan opsi-opsinya, memudahkan pembacaan dan pemahaman detail pesanan. Implementasi ini membuat konsistensi tampilan struk di semua jenis printer yang didukung oleh sistem, baik *Bluetooth printer*, *Verifone printer*, maupun *Sunmi printer*, sehingga pengalaman pengguna tetap sama.




Gambar 3.56. Struk transaksi pada printer sunmi dengan *optionItem*

Kemudian pada gambar 3.56 merupakan lampiran dari hasil struk yang memiliki opsi pada printer sunmi. Format masih tetap sama dengan jenis perangkat printer lainnya

## 3.4 Kendala dan Solusi yang Ditemukan

Selama pelaksanaan program magang kerja di MID, penulis terkadang menghadapi beberapa kendala yang pada akhirnya dapat ditemukan jawaban dan solusinya. Beberapa kendala tersebut diantaranya ialah:

1. Kendala teknis ketika proyek tidak dapat dijalankan saat pertama kali memulai ke dalam proyek ini. Proyek ini memiliki beberapa library yang sudah *deprecated* atau tidak tersedia lagi di *Jcenter*, terutama dependensi sunmidslib yang tidak ditemukan di repository manapun.
2. Proyek aplikasi ini, sebelumnya terintegrasi dengan *backend* versi Node.js, sehingga ketika pertama kali masuk ke proyek ini, hampir semua fungsi yang menggunakan *API* tidak dapat berjalan. Aplikasi harus dilakukan penyesuaian dengan struktur *backend API* yang baru yaitu golang.

3. Ketika ditugaskan untuk membuat socket server dengan *Node.js* yang saling berkomunikasi dengan gRPC, penulis mengalami kendala saat pengembangan dilakukan. Karena ini merupakan teknologi yang baru diketahui oleh penulis.

Dari beberapa kendala di atas, dapat ditemukan solusi seperti berikut:

1. Dilakukan pencarian alternatif dari beberapa dependensi yang sudah tidak tersedia di repository yang masih aktif, seperti Maven Central, Github, dan jitpack.io. Untuk dependensi sunmidslib yang tidak ditemukan, dilakukan penonaktifan sementara bagian kode yang menggunakan dependensi tersebut agar proyek dapat dijalankan.
2. Dilakukan pembelajaran terlebih dahulu terhadap dokumentasi API backend Golang yang baru, kemudian dilakukan penyesuaian endpoint dan format request/response, serta testing pada fungsi yang baru disesuaikan tersebut.
3. Diberikan tugas untuk mempelajari terlebih dahulu terkait teknologi gRPC dan pembuatan socket server. Supervisor juga memberikan arahan terkait pengembangan teknologi tersebut.