BAB 3
PELAKSANAAN KERJA MAGANG

3.1 Kedudukan dan Koordinasi

Pelaksanaan kerja magang pada bagian *IT Developer* di bawah Departemen Teknologi Informasi sebagai *Developer Intern*. Selama kegiatan magang, bimbingan diberikan oleh *Head of IT Department* selaku mentor yang secara aktif memberikan arahan dan masukan dalam proses pengembangan. Tugas utama yang dilaksanakan mencakup pengembangan sistem, pengelolaan kode melalui *GitHub* untuk mempermudah kolaborasi dan pemisahan fitur, serta pemecahan kendala teknis yang muncul selama proses.

Koordinasi dilakukan melalui rapat rutin dan diskusi langsung untuk menyampaikan progres serta memperoleh *feedback*, dengan evaluasi dilakukan secara berkala. Selain itu, rapat mingguan bersama *Head of IT Department* dilakukan untuk membahas perkembangan proyek dan permasalahan teknis yang dihadapi.

3.2 Tugas yang Dilakukan

Selama melaksanakan magang di PT Mobile Data Indonesia pada bagian *IT Developer*, tugas diberikan untuk membuat dan mengembangkan sistem yang berfokus pada kelancaran operasional perusahaan. Tugas utama yang dilakukan adalah sebagai berikut:

1. Mengimplementasikan sistem *Client Portal* dengan tujuan memudahkan klien dalam memperoleh informasi.
2. Membuat pengingat melalui email untuk klien apabila mendekati tenggat waktu tertentu terkait pembayaran invoice bulanan.
3. Melakukan pemeliharaan sistem untuk memastikan sistem berjalan dengan lancar dan stabil.

3.3 Uraian Pelaksanaan Magang

Kegiatan magang di PT Mobile Data Indonesia dilaksanakan selama enam bulan sesuai dengan ketentuan kontrak magang. Program magang dimulai pada

tanggal 26 Agustus 2025 dan berakhir pada tanggal 28 Februari 2026. Pelaksanaan magang dilakukan pada hari kerja sesuai dengan jadwal operasional perusahaan.

Tabel 3.1 menyajikan rincian linimasa kegiatan yang dilaksanakan setiap minggunya selama periode magang di PT Mobile Data Indonesia.

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

Minggu Ke -	Pekerjaan yang dilakukan
1	Kegiatan dimulai dengan rapat awal untuk memperkenalkan proyek, diikuti dengan mempelajari sistem yang telah ada sebelumnya beserta fungsinya, seperti membaca dokumentasi frontend, membaca API contract dan melakukan testing API contract, mempelajari flow chart, serta berdiskusi dan bertanya kepada karyawan lainnya untuk memperdalam pemahaman terhadap sistem.
2	Kegiatan difokuskan pada pembelajaran dasar bahasa pemrograman Go (Golang) sebagai fondasi proyek, mencakup konsep-konsep seperti *slice*, *array*, konversi tipe data, penggunaan *defer*, *panic*, *alert*, serta pemahaman terhadap *interface* dan *struct*
3	Kegiatan difokuskan pada pembelajaran Golang terkait modul, mekanisme *import* dan *export*, serta penerapan *unit test* menggunakan module. Selain itu, dilakukan pemahaman terhadap *framework* Fiber, termasuk konsep *routing* dan penggunaan *context*.
4	Pada minggu ini dilakukan modifikasi struktur database dengan penambahan dan kolom baru sesuai kebutuhan pengembangan fitur. Pembuatan dan pengujian data seeder untuk client dilakukan guna mendukung proses pengembangan dan pengujian berikutnya. Implementasi autentikasi menggunakan Next Auth dikembangkan mulai dari konfigurasi dasar, pembuatan endpoint login, hingga penyempurnaan flow autentikasi.
	Lanjut pada halaman berikutnya

Tabel 3.1 Pekerjaan yang dilakukan tiap minggu selama pelaksanaan kerja magang (lanjutan)

Minggu Ke -	Pekerjaan yang dilakukan
5	Pembuatan endpoint untuk mendapatkan seluruh data invoice dari database dilakukan, termasuk penyusunan struktur response dan penanganan error. Endpoint untuk pengambilan data payment juga dikembangkan dengan penyesuaian query parameter dan filter data yang diperlukan. Penerapan Design halaman login pada frontend dilakukan berdasarkan desain yang telah tersedia, diikuti dengan implementasi form login dan integrasinya dengan endpoint backend.
6	Pembuatan navbar responsif dilakukan untuk mendukung tampilan aplikasi pada berbagai ukuran perangkat. Halaman untuk menampilkan seluruh data invoice diimplementasikan dengan memanfaatkan endpoint yang telah tersedia. Proses migrasi dari axios ke useSWR dilakukan untuk meningkatkan efisiensi data fetching dan caching.
7	Pembelajaran dan eksplorasi dilakukan terhadap Puppeteer Library untuk keperluan generate PDF, mencakup fitur dasar serta konfigurasi awal. Proses dilanjutkan dengan percobaan generate dan download PDF menggunakan Puppeteer sebelum akhirnya beralih ke React PDF setelah evaluasi kebutuhan proyek.
8	Rework dilakukan pada endpoint payment dengan penambahan fungsi untuk membaca gambar dalam format blob sebagai base64. Perbaikan dilakukan terhadap error yang muncul akibat blob yang tidak terdeteksi atau gagal didekompilasi. Halaman details payment dikembangkan untuk menampilkan informasi pembayaran secara lengkap, diikuti dengan implementasi view payment endpoint yang terintegrasi dengan data detail dan gambar base64.
9	Pembuatan cron job dilakukan untuk mengirim email reminder secara otomatis sesuai jadwal yang telah ditentukan. Pengembangan sistem email notification dilanjutkan dengan penyempurnaan mekanisme pengiriman dan penanganan error. Template email notification dirancang dan dibuat agar selaras dengan kebutuhan serta identitas perusahaan, kemudian diintegrasikan ke dalam sistem pengiriman dan diuji secara awal.
	Lanjut pada halaman berikutnya

Tabel 3.1 Pekerjaan yang dilakukan tiap minggu selama pelaksanaan kerja magang (lanjutan)

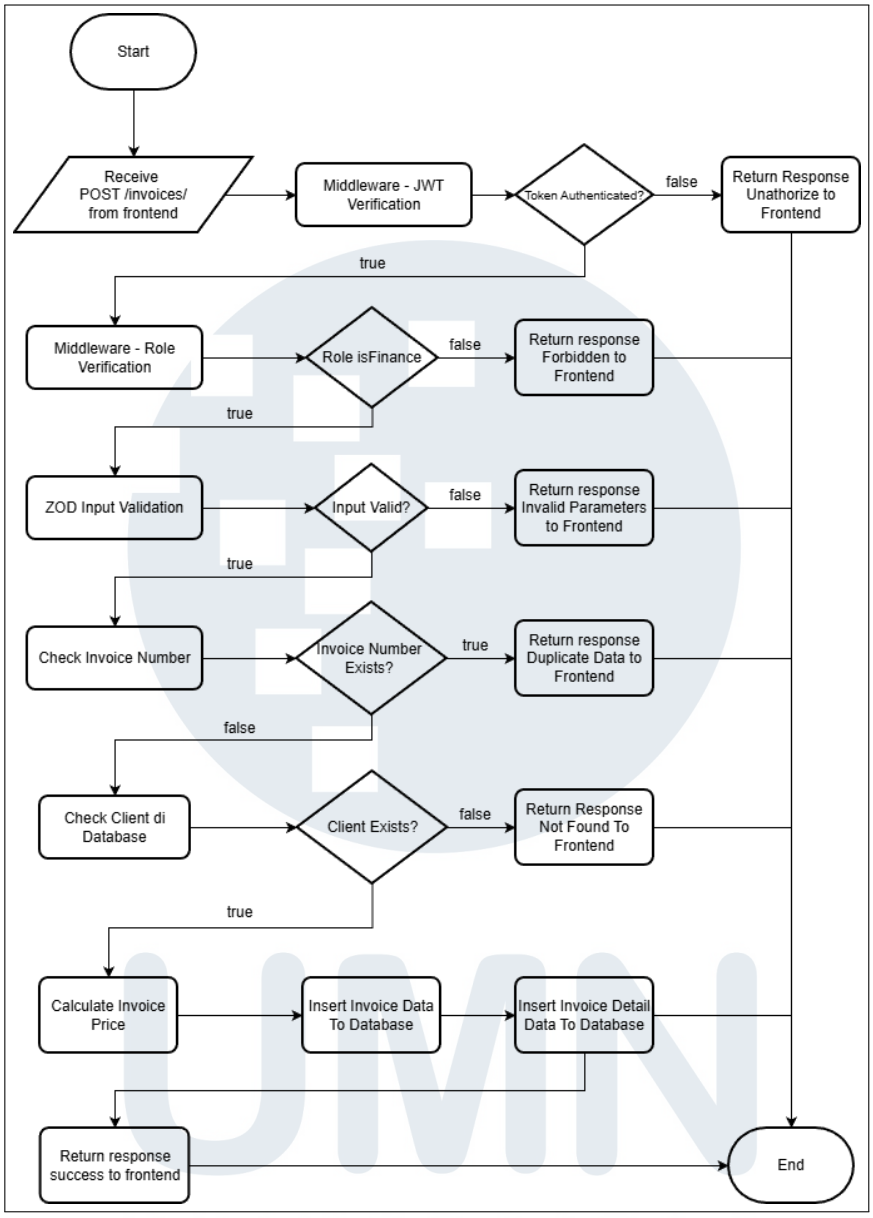
Minggu Ke -	Pekerjaan yang dilakukan
10	Implementasi desain dilakukan pada halaman invoice agar sesuai dengan desain dan standar visual yang telah ditetapkan perusahaan. Halaman payment juga disesuaikan agar konsisten dengan pedoman desain dan antarmuka lainnya. Pengembangan dashboard utama dimulai dengan perancangan layout serta penyusunan komponen inti, kemudian dilanjutkan dengan implementasi grafik dan elemen interaktif sesuai kebutuhan. Responsivitas diterapkan pada dashboard utama untuk mendukung tampilan yang optimal di berbagai perangkat.
11	Integrasi *display* nomor *invoice* (*Invoice_Number*) pada antarmuka *Payment* dan *Invoice* diselesaikan untuk optimalisasi pelacakan data. Kapabilitas *filtering* ditingkatkan melalui implementasi sistem *filter* berbasis status dan tanggal pada Tabel *Invoice* dan sistem *filter* kriteria kompleks pada Tabel *Payment*, dengan penyesuaian tampilan agar responsif. *Conditional formatting* diterapkan pada Tabel *Invoice* untuk visualisasi status kritis (*invoice* melewati jatuh tempo). Terakhir, fitur notifikasi *email* otomatis diimplementasikan untuk pengiriman pengingat saat *invoice* melewati batas waktu pembayaran (*deadline*).
12	Aspek keamanan sistem ditingkatkan melalui implementasi *rate limiter* pada aplikasi *backend* dan penambahan fungsi *redirect* 404 untuk penanganan halaman yang tidak ditemukan. Pada sisi *frontend*, *protected routes* diimplementasikan. Pengembangan sistem notifikasi, komponen tampilan notifikasi pada *frontend* dikembangkan dengan memastikan desain yang responsif. Fungsionalitas *cron job* yang sudah ada diperluas dan dioptimalkan untuk membuat notifikasi otomatis bagi *invoice* yang belum dibayar dan *invoice* yang telah melewati masa pembayaran.
13	Fokus utama periode ini adalah *seeding data* dalam volume besar diimplementasikan guna mendukung skenario simulasi dan *load testing*. Aktivitas ini dilanjutkan dengan pelaksanaan *testing* terhadap seluruh fitur yang tersedia untuk memastikan fungsionalitas sistem.

3.3.1 Mempelajari Sistem Sebelumnya

Pada minggu pertama kegiatan magang, dilakukan penjelasan mengenai kelanjutan program yang telah dikembangkan sebelumnya. Untuk memperoleh pemahaman yang lebih mendalam terhadap program tersebut, dilakukan analisis *screening* secara cermat dan efisien terhadap sistem yang sudah ada. Dalam konteks ini, *flowchart* merupakan alat yang krusial, sebab *flowchart* adalah representasi diagramatik dari langkah-langkah suatu algoritma [5, 6]. Adapun cuplikan *flowchart* dan *API Contract* dari aplikasi yang ada sebelumnya disajikan sebagai berikut:

A Flowchart Pembuatan Invoice

Diagram alir pada Gambar 3.1[3] mengilustrasikan prosedur penanganan permintaan `POST /invoices/`. Proses dimulai dengan `Middleware - JWT Verification` untuk otentikasi, di mana *error* akan mengembalikan respons `Unauthorized`. Validasi dilanjutkan dengan `Role Verification` (memastikan peran `isFinance`) dan `ZOD Input Validation` pada tahap ini masing-masing mengembalikan respons `Forbidden` dan `Invalid Parameters`. Selanjutnya, sistem melakukan pemeriksaan duplikasi nomor *invoice* yang jika ditemukan akan mengembalikan respons `Duplicate Data`. Proses juga mencakup verifikasi keberadaan klien, yang kegagalannya direspon dengan `Not Found`. Hanya setelah seluruh validasi berhasil, sistem melanjutkan ke eksekusi utama: `Calculate Invoice Price`, diikuti dengan penyisipan data ke basis data `Invoice` dan `Invoice Detail`. Prosedur diselesaikan dengan pengembalian respons `success to frontend` dan terminasi (`End`).

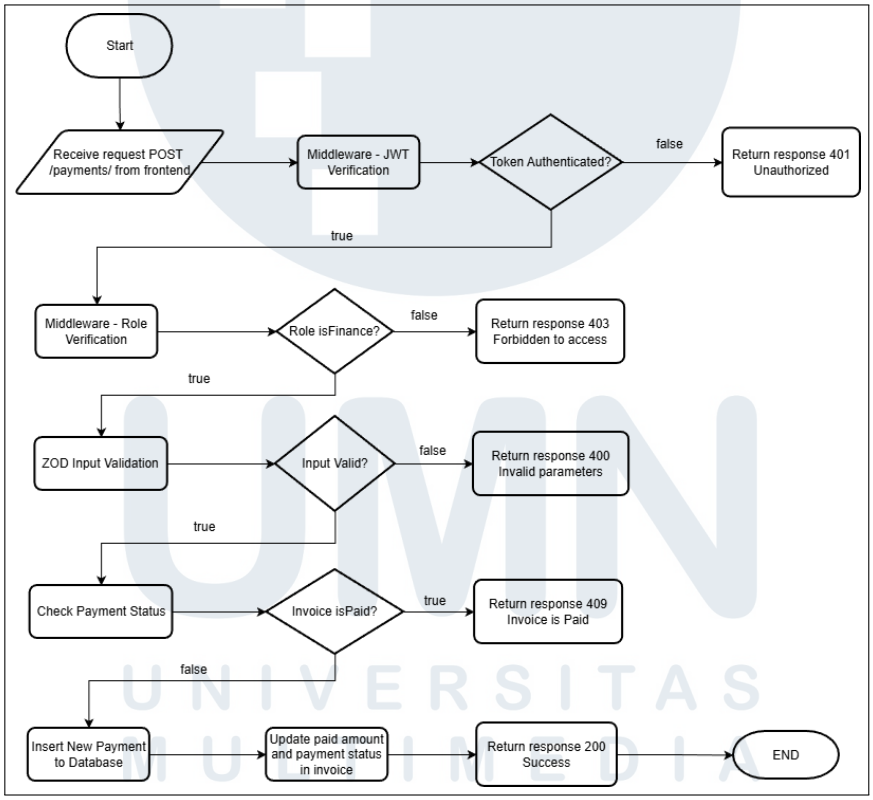


Gambar 3.1. Flowchart pembuatan invoice baru

B Flowchart Pembuatan Payment

Diagram alir pada Gambar 3.2 [3] menyajikan alur pemrosesan permintaan pembayaran (`POST /payments/`) dari antarmuka pengguna. Prosedur ini diinisiasi dengan penerimaan permintaan lalu dilanjutkan melalui serangkaian validasi otorisasi dan integritas data yang ketat. Tahap pertama mencakup verifikasi token `JWT` terhadap basis data `Users`. Kegagalan otentikasi akan menghentikan proses dicatat dan mengembalikan respons `401 Unauthorized`. Jika otentikasi

berhasil sistem melanjutkan ke verifikasi peran (`Role Verification`). Kegagalan verifikasi peran direspon dengan pencatatan dan respons `403 Forbidden`. Validasi selanjutnya adalah `Input Validation` untuk memastikan integritas data masukan. Kegagalan validasi ini direspons dengan `400 Invalid parameters`. Setelah validasi input sistem melakukan pemeriksaan status pembayaran pada `Invoice` terkait. Jika status pembayaran sudah `Paid` transaksi dicegah dicatat dan dihentikan dengan respons `409 Invoice is Paid`. Apabila semua verifikasi lolos proses berlanjut ke eksekusi transaksi. Eksekusi ini mencakup penyisipan pembayaran baru ke basis data `Payments` dan pembaruan status serta jumlah yang telah dibayar pada basis data `Invoices`. Sebagai tahap terminasi yang sukses aktivitas ini dicatat dan sistem mengembalikan respons `200 Success` sebelum prosedur berakhir (`END`).



Gambar 3.2. Flowchart pembuatan pembayaran baru

C API Contract Pembuatan Invoice

Kontrak API ini mendefinisikan mekanisme untuk membuat sebuah *Invoice* baru beserta detailnya melalui metode `POST` pada *endpoint* `/invoices`. API ini memerlukan otentikasi menggunakan token.

C.1 Request Body

Request body wajib disertakan dalam format `application/json`. Bagian ini memuat data utama *invoice* serta rincian *item* di dalamnya. Struktur data lengkap terkait *request body* tersebut dipaparkan pada Tabel 3.2 berikut ini:

Tabel 3.2. Struktur data request API pembuatan invoice

Parameter	**Tipe Data**
Data Invoice Utama	
`invoice_number`	`string`
`issue_date`	`string` (*date*)
`due_date`	`string` (*date*)
`tax_rate`	`number`
`tax_invoice_number`	`string`
`client_id`	`string` (*uuid*)
Detail Invoice	
`transaction_note`	`string`
`delivery_count`	`number`
`price_per_delivery`	`number`

C.2 Response

API ini memberikan beberapa kode status respons HTTP:

- `201 Created`: Invoice berhasil dibuat.
- `400 Bad Request`: Parameter yang dikirim tidak valid (kesalahan validasi data).
- `404 Not Found`: Klien tidak ditemukan berdasarkan `client_id`.
- `409 Conflict`: Invoice sudah ada.
- `500 Internal Server Error`: Kesalahan server internal.

C.3 Struktur Respons Berhasil

Respons sukses menyajikan data *invoice* yang telah diproses, mencakup nilai kalkulasi seperti `sub total`, `tax amount`, dan `total`. Detail mengenai struktur data tersebut dapat dirujuk pada Tabel 3.3 sebagai berikut:

Tabel 3.3. Struktur data response sukses pembuatan invoice

Parameter	**Tipe Data**
`status`	`string`
`code`	`integer`
`message`	`string`
Data	
`invoice_id`	`string` (*uuid*)
`invoice_number`	`string`
`issue_date`	`string` (*date-time*)
`due_date`	`string` (*date-time*)
`sub_total`	`number`
`tax_rate`	`number`
`tax_amount`	`number`
`total`	`number`
`tax_invoice_number`	`string`
`amount_paid`	`number`
`payment_status`	`string` (enum: `[paid, unpaid, partial]`)
`voided_at`	`string` (*date-time*, *nullable*)
`client_id`	`string` (*uuid*)
`created_at`	`string` (*date-time*)
`updated_at`	`string` (*date-time*)

C.4 Struktur Respons Error

Respons kesalahan memiliki skema umum yang mencakup status, kode, pesan, dan data, di mana bagian data dapat memuat rincian validasi atau bernilai `null`. Penjelasan mengenai komponen struktur tersebut dipaparkan pada Tabel 3.4 sebagai berikut:

Tabel 3.4. Struktur data response error pembuatan invoice

Parameter	**Tipe Data**
`status`	`string`
`code`	`integer`
`message`	`string`
`data`	`object` atau `null`

D API Contract Pembuatan Payment

Kontrak API ini mendefinisikan mekanisme untuk mencatat sebuah Pembayaran (Payment) baru terhadap *invoice* tertentu melalui metode `POST`

pada *endpoint* `/payments`. API ini memerlukan otentikasi menggunakan token dan menerima data dalam format `multipart/form-data` karena melibatkan pengunggahan berkas bukti transfer.

D.1 Request Body

Penggunaan *request body* diwajibkan menggunakan format `multipart/form-data` untuk memastikan validitas pengiriman data. Spesifikasi mengenai parameter dan struktur data yang diperlukan dalam proses ini dipaparkan pada Tabel 3.5 berikut:

Tabel 3.5. Struktur data request API pembuatan pembayaran

Parameter	**Tipe Data**
`payment_date`	`string` (*date*)
`amount_paid`	`number`
`proof_of_transfer`	`string`
`invoice_number`	`string`

D.2 Response

API ini memberikan beberapa kode status respons HTTP:

- `201 Created`: Pembayaran berhasil dibuat. Respons mengembalikan objek pembayaran yang baru dibuat.
- `400 Bad Request`: Parameter yang dikirim tidak valid (kesalahan validasi data).
- `404 Not Found`: Invoice tidak ditemukan berdasarkan `invoice_number`.
- `409 Conflict`: Konflik pembayaran.
- `500 Internal Server Error`: Kesalahan server internal.

D.3 Struktur Respons Berhasil

Respons sukses akan mengembalikan struktur standar dan objek data pembayaran yang tercipta.

Tabel 3.6. Struktur data response sukses pembuatan pembayaran

Parameter	Tipe Data
`status`	`string`
`code`	`integer`
`message`	`string`
`data`	`object`

D.4 Struktur Respons Error

Respons kesalahan mengadopsi struktur standar yang mencakup elemen status, kode, pesan, serta data, yang dapat berisi rincian validasi maupun bernilai `null`. Adapun spesifikasi mengenai komponen struktur tersebut dipaparkan pada Tabel 3.7 berikut ini:

Tabel 3.7. Struktur data response error pembuatan pembayaran

Parameter	Tipe Data
`status`	`string`
`code`	`integer`
`message`	`string`
`data`	`object` atau `null`

3.3.2 Belajar Go

Pada minggu ke-2 dan ke-3, kegiatan difokuskan pada pembelajaran bahasa pemrograman Go (*Golang*) sebagai fondasi teknis untuk pengembangan *backend* menggunakan *framework* Go Fiber. Pembelajaran mencakup konsep-konsep fundamental seperti tipe data dasar, struktur data kolektif (*slice*, *array*, *map*), manajemen kontrol alur (*defer*, *panic*), hingga pemahaman mendalam tentang *struct* dan *interface* untuk membangun arsitektur modular. Selain itu, dipelajari mekanisme *import/export* modul, penerapan *unit test*, dan konsep *routing* serta *context* dalam *framework* Fiber.

A Tipe Data dan Slice

Pembelajaran fundamental mencakup deklarasi variabel, konversi tipe data, serta penggunaan struktur data kolektif dinamis seperti *slice* dan *map*. hasil pembelajara

```go
var suhuCelcius float32 = 27.5
var suhuFahrenheit float64 = float64(suhuCelcius * 9/5 + 32)
var nilaiInt int16 = int16(suhuCelcius) // Konversi ke integer

type KodeProduk string
var kodeBarang KodeProduk = "PRD-2025-A1"
```

Kode 3.1: Deklarasi variabel dan konversi tipe data

Kode 3.1 mengilustrasikan konversi eksplisit antar tipe data numerik. Selain itu, Go mendukung deklarasi tipe data baru berdasarkan tipe data yang sudah ada (*type aliases*), seperti `KodeProduk` yang didasarkan pada tipe `string`.

```go
dataPegawai := [...]string{"Rina", "Agus", "Hendra", "Maya", "Siska", "Arief"}
fmt.Println(dataPegawai[2:5]) // Hasil: [Hendra Maya Siska]

arsipInisiatif := make([]string, 3, 8)
arsipInisiatif[0] = "Inisiatif Alpha"
arsipInisiatif[1] = "Inisiatif Beta"
arsipInisiatif[2] = "Inisiatif Gamma"
arsipInisiatifTerbaru := append(arsipInisiatif, "Inisiatif Delta")
```

Kode 3.2: Contoh inisialisasi dan manipulasi slice

Kode 3.2 menunjukkan penggunaan *slice* dalam bahasa Go, yang meliputi pengambilan sebagian elemen menggunakan teknik *slicing*, pembuatan *slice* dengan fungsi bawaan `make` yang memiliki panjang dan kapasitas tertentu, serta penambahan elemen baru ke dalam *slice* menggunakan fungsi `append`.

B Struktur Data dan Interface

Pembelajaran dilanjutkan pada struktur data kustom (*struct*) untuk pemodelan data dan *interface* untuk implementasi polimorfisme. Kode 3.3 mendefinisikan *struct* `Pegawai` yang memiliki properti dasar. Fungsi `getSapaanKepada` adalah *method* yang terikat pada *struct* `Pegawai`, memungkinkan *struct* tersebut memiliki perilaku.

```go
type Pegawai struct{
  Nama, Jabatan string
  IDPegawai    int
}

```

```
func (p Pegawai) getSapaanKepada(target string) (result string){
  result = "Selamat siang, " + target + ". Saya " + p.Nama + ",
    dari bagian " + p.Jabatan
  return result
}
```

Kode 3.3: Definisi struct dan method

Kode 3.4 menunjukkan definisi *interface* `Laporan`. Fungsi `cetakLaporan` dapat menerima objek apa pun (seperti *struct* `Pegawai`) selama objek tersebut mengimplementasikan *method* `BuatLaporan` secara implisit.

```
type Laporan interface{
  BuatLaporan(judul string) string
}

func cetakLaporan(objek Laporan){
  fmt.Println(objek.BuatLaporan("Laporan Bulanan"))
}
// Struct Pegawai harus mengimplementasikan method BuatLaporan
func (p Pegawai) BuatLaporan(judul string) (string){
// ...
}
```

Kode 3.4: Interface dan penerapannya

C Manajemen Kontrol Alur Defer, Panic, dan Recover

Manajemen galat dalam bahasa pemrograman Go memanfaatkan mekanisme *defer*, *panic*, dan *recover* untuk mengendalikan alur eksekusi program saat terjadi kondisi abnormal. Kode 3.5 memperlihatkan bagian-bagian inti dari implementasi mekanisme tersebut.

```
func ujiDeferPanic(triggerPanic bool) {
  defer func() {
    if pesan := recover(); pesan != nil {
      fmt.Println("Error:", pesan)
    }
  }()

  if triggerPanic {
    panic("Koneksi Database Gagal")
  }
}
```

Kode 3.5: Contoh inti penggunaan defer, panic, dan recover

Cuplikan kode tersebut menyoroti peran utama `defer` dalam menjamin eksekusi fungsi pemulihan ketika terjadi *panic*. Pemanggilan `panic` digunakan untuk mensimulasikan kegagalan sistem, sementara fungsi `recover()` yang dieksekusi di dalam blok `defer` berfungsi untuk menangkap dan mengelola galat tersebut. Penyederhanaan kode ini bertujuan untuk menekankan konsep fundamental tanpa menampilkan detail implementasi yang tidak relevan dengan pembahasan.

D Error Handling

Dalam bahasa pemrograman Go, penanganan galat (*error handling*) umumnya dilakukan dengan mengembalikan nilai bertipe `error` sebagai nilai kembalian terakhir dari suatu fungsi. Kode 3.6 memperlihatkan penerapan mekanisme tersebut, termasuk pembuatan tipe galat kustom dan pengembalian nilai `nil` ketika proses berjalan tanpa kesalahan.

```
type InputError struct {
  Pesan string
}

func (e *InputError) Error() string {
  return e.Pesan
}

func prosesData(isError bool) (string, error) {
  if isError {
    return "", errors.New("Validasi data input gagal")
  }
  return "Data berhasil diproses", nil
}
```

Kode 3.6: Fungsi dengan error handling standar

Pada cuplikan kode tersebut, tipe galat kustom `InputError` didefinisikan dengan mengimplementasikan *method* `Error()`, sehingga memenuhi antarmuka `error` yang disediakan oleh Go. Selain itu, fungsi `prosesData` mengilustrasikan praktik konvensional dalam Go, yaitu mengembalikan nilai `nil` sebagai *error* apabila proses eksekusi berhasil, serta objek *error* ketika terjadi kegagalan.

E Implementasi Sederhana Go Fiber Routing dan Handler

Pembelajaran *framework* Go Fiber difokuskan pada konsep *routing* serta pemanfaatan *context* untuk menangani permintaan HTTP. Kode 3.7 menunjukkan contoh implementasi dasar sebuah *endpoint* menggunakan metode HTTP `GET` yang mengembalikan respons dalam format JSON.

```
package main

import (
  "log"
  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()

  app.Get("/api/status", func(c *fiber.Ctx) error {
    return c.Status(fiber.StatusOK).JSON(fiber.Map{
      "message": "Aplikasi siap digunakan",
      "status":  "OK",
    })
  })

  log.Fatal(app.Listen(":3000"))
}
```

Kode 3.7: Inisialisasi aplikasi dan endpoint sederhana

Pada implementasi tersebut, aplikasi Fiber diinisialisasi melalui pemanggilan fungsi `fiber.New()`. Selanjutnya, metode `app.Get` digunakan untuk mendefinisikan sebuah *route* dengan metode HTTP `GET` pada jalur `/api/status`. Fungsi *handler* menerima parameter bertipe `*fiber.Ctx` yang merepresentasikan konteks permintaan, dan digunakan untuk mengatur kode status HTTP serta mengirimkan respons dalam format JSON kepada klien. Aplikasi kemudian dijalankan sebagai server HTTP pada *port* 3000.

3.3.3 Modifikasi Struktur Tabel untuk Integrasi Kredensial

Pada minggu keempat, pengembangan diawali dengan melakukan perubahan pada struktur model `Client`. Modifikasi ini bertujuan mengubah tabel yang sebelumnya hanya menyimpan profil klien menjadi tabel yang juga berfungsi

sebagai penyimpanan kredensial untuk proses otentikasi. Perubahan dilakukan dengan menambahkan kolom `client_email` dan `client_password`. Perbandingan struktur sebelum dan sesudah perubahan ditampilkan pada Tabel 3.8 dan Tabel 3.9.

Tabel 3.8. Struktur model client (sebelum)

Atribut	**Tipe Data**
client_id	String
client_name	String
currency	String
country	String
client_address	String
postal_code	String
client_phone	String
deleted_at	DateTime?
created_at	DateTime
updated_at	DateTime

Tabel 3.9. Struktur model client (sesudah)

Atribut	**Tipe Data**
client_id	String
client_name	String
client_email	String
client_password	String
currency	String
country	String
client_address	String
postal_code	String
client_phone	String
deleted_at	DateTime?
created_at	DateTime
updated_at	DateTime

3.3.4 Implementasi Seeder untuk Pengujian

Pada tahap pengembangan API, dibuat dua *database seeder* untuk menyediakan data awal selama proses pengujian, yaitu *Client Seeder* dan *Invoice Seeder*. Seeder digunakan agar pengujian API dapat berjalan secara konsisten dengan data yang stabil.

A Client Seeder

Client Seeder digunakan untuk menyediakan beberapa entitas klien awal yang dapat digunakan untuk proses autentikasi. Password klien di-*hash* menggunakan `bcrypt` agar sesuai dengan praktik keamanan.

```
hashedPassword, _ := bcrypt.GenerateFromPassword([]byte("
    password123"), bcrypt.DefaultCost)
db.Exec(`
  INSERT INTO clients (client_id, client_email, client_password,
    client_name)
  VALUES (gen_random_uuid(), ?, ?, ?)`,
  "contact@nusantaratech.co.id", string(hashedPassword), "
    nusantaratech",
)
```

Kode 3.8: Contoh hashing dan insert klien

Kode di atas menunjukkan proses inti dari Client Seeder: hashing password dan penyisipan data klien secara langsung. Pada implementasi sebenarnya, beberapa klien ditambahkan secara *batch insert*, namun ditampilkan satu contoh untuk menjaga kejelasan.

B Invoice Seeder

Invoice Seeder menghasilkan data faktur dengan nilai perhitungan otomatis seperti subtotal, pajak, dan total. Penyisipan data dilakukan dalam sebuah transaksi untuk memastikan integritas antara tabel `invoices` dan `invoice_details`.

```
db.Transaction(func(tx *gorm.DB) error {
    // Hitung nilai faktur
    price := 100
    count := 2000
    subTotal := price * count
    taxAmount := 0.02 * float64(subTotal)
```

```
    total := float64(subTotal) + taxAmount

    // Insert invoice dan ambil invoice_id
    var invoiceID string
    tx.Raw(`
      INSERT INTO invoices (
        invoice_id, invoice_number, issue_date, due_date,
        tax_rate, tax_amount, sub_total, total, payment_status
      )
      VALUES (gen_random_uuid(), ?, NOW(), NOW() + INTERVAL '30 day',
      ?, ?, ?, ?, 'unpaid')
      RETURNING invoice_id`,
      "INV-001", 0.02, taxAmount, subTotal, total,
    ).Scan(&invoiceID)

    // Insert invoice detail
    tx.Exec(`
      INSERT INTO invoice_details (
        invoice_detail_id, invoice_id, amount, price_per_delivery, delivery_count
      )
      VALUES (gen_random_uuid(), ?, ?, ?, ?)`,
      invoiceID, subTotal, price, count,
    )

    return nil
})
```

Kode 3.9: Contoh transaksi invoice seeder

Transaksi di atas memastikan bahwa data faktur dan detail faktur hanya akan disimpan jika kedua operasi berhasil. Dengan demikian, konsistensi data tetap terjaga selama proses pengujian.

3.3.5 Implementasi Fitur Autentikasi Klien

Sistem autentikasi dibangun dengan memisahkan tanggung jawab antara manajemen permintaan, logika validasi keamanan, dan akses data untuk memenuhi prinsip *separation of concerns*.

Pada lapisan *handler*, fokus utama adalah mendelegasikan data kredensial dari permintaan HTTP ke lapisan layanan. Implementasi secara selektif untuk

proses delegasi ini dapat dilihat pada Kode 3.10.

```
func (h *ClientHandler) LoginHandler(c *fiber.Ctx) error {
    // ... proses parsing request ...
    client, err := h.service.LoginService(req.Email, req.Password)
    // ... penanganan error dan response ...
    return utils.Success(c, 200, "Login success", client)
}
```

Kode 3.10: Handler autentikasi

Logika bisnis utama terdapat pada lapisan *service*, khususnya pada proses verifikasi keamanan menggunakan algoritma *hashing*. Inti dari validasi kredensial pengguna tersebut dijelaskan pada Kode 3.11.

```
func (cs *clientService) LoginService(email, password string)
    (...) {
    client, _ := cs.repo.FindByEmail(email)

    // Inti verifikasi: Komparasi hash password menggunakan bcrypt
    if err := bcrypt.CompareHashAndPassword([]byte(client.
    ClientPassword), []byte(password)); err != nil {
        return models.ClientLoginResponse{}, errors.New("password
    salah")
    }
    return utils.ToClientLoginResponse(client), nil
}
```

Kode 3.11: Inti logika verifikasi service

Interaksi dengan basis data pada lapisan *repository* disederhanakan untuk hanya melakukan pengambilan entitas tunggal. Kueri data berdasarkan identitas unik surel ditunjukkan pada Kode 3.12.

```
func (r *clientRepository) FindByEmail(email string) (*models.
    Client, error) {
    var client models.Client
    // Pengambilan data menggunakan kueri SQL mentah
    err := r.db.Raw(`SELECT * FROM clients WHERE client_email = ?
    LIMIT 1`, email).Scan(&client).Error
    return &client, err
}
```

Kode 3.12: Kueri data klien

3.3.6 Integrasi Autentikasi Sisi Frontend

Di sisi *frontend*, integrasi dilakukan untuk menghubungkan *input* pengguna dengan *endpoint* API *backend*. Logika utama dalam pemberian otorisasi sesi pengguna pada pustaka NextAuth dirinci pada Kode 3.13.

```
authorize: async (cred) => {
  // Komunikasi asinkronus dengan backend API
  const res = await axios.post(`${process.env.API_URL}/api/clients
   /login`, cred);
  const data = res.data.data;

  // Mengembalikan data user jika autentikasi berhasil
  return data ? { id: data.client_id, email: data.client_email,
   name: data.client_name } : null;
}
```

Kode 3.13: Logika authorize NextAuth

Melalui pendekatan yang dipaparkan pada Kode 3.10 hingga Kode 3.13, aplikasi menjamin bahwa hanya pengguna dengan kredensial valid yang dapat memperoleh token akses untuk sesi aktif.

3.3.7 Implementasi Fitur Pengambilan Data Invoice

Sistem ini memastikan isolasi data dengan mengimplementasikan alur kerja berlapis. Fokus utama pada fitur ini adalah memastikan bahwa data yang ditarik hanya milik pengguna yang terautentikasi.

A Lapisan Handler

Pada lapisan ini, poin krusial adalah pengambilan identitas klien dari konteks lokal yang telah diatur oleh *middleware*. Implementasi inti pada `GetAllInvoiceByClientIdHandler` dapat dilihat pada Kode 3.14.

```
// Mengambil identitas user dari context yang diinjeksi middleware
userId := c.Locals("userId").(string)
invoices, err := h.service.GetAllInvoiceByClientIdService(userId)
```

Kode 3.14: Ekstraksi ID dari konteks lokal

B Lapisan Service

Lapisan *service* berfungsi melakukan validasi bisnis terhadap hasil kueri. Bagian paling penting adalah deteksi kondisi data yang tidak ditemukan untuk memberikan respons spesifik, seperti ditunjukkan pada Kode 3.15.

```
// Validasi spesifik jika record tidak ditemukan dalam database
if errors.Is(err, gorm.ErrRecordNotFound) {
    return nil, errors.New("invoice tidak ditemukan")
}
```

Kode 3.15: Logika validasi record

C Lapisan Repository dan Struktur Data

Pada lapisan *repository*, keamanan data dijamin melalui penggunaan kueri SQL yang terfilter secara eksplisit. Fokus utama terletak pada implementasi klausa `WHERE` menggunakan atribut `client_id` untuk membatasi ruang lingkup pengambilan data, sehingga tercipta isolasi data antar klien.

Penyimpanan data faktur diorganisir ke dalam dua entitas utama: tabel `invoices` untuk informasi *header* dan tabel `invoice_details` untuk rincian transaksi. Hubungan antar tabel ini dijaga melalui *foreign key* `invoice_id`. Struktur lengkap dari kedua tabel tersebut dipaparkan pada Tabel 3.12 dan Tabel 3.11.

Tabel 3.10. Struktur table invoice

Atribut	**Tipe Data**
invoice_id	String (PK)
invoice_number	String
issue_date	DateTime
due_date	DateTime
total	Float
payment_status	String
client_id	String (FK)
created_at	DateTime

Tabel 3.11. Struktur table invoice detail

Atribut	Tipe Data
invoice_detail_id	String (PK)
transaction_note	String
amount	Float
invoice_id	String (FK)
created_at	DateTime

Berdasarkan struktur pada Tabel 3.12, kolom `client_id` menjadi kunci krusial dalam keamanan aplikasi. Setiap kueri yang dieksekusi oleh *repository* wajib menyertakan filter terhadap kolom ini untuk memastikan klien hanya dapat mengakses data miliknya sendiri. Implementasi kueri tersebut ditunjukkan pada Kode 3.16.

```
// Menjamin isolasi data melalui filter client_id pada kueri
query := 'SELECT * FROM invoices WHERE client_id = ?'
err := r.db.Raw(query, clientId).Scan(&invoices).Error
```

Kode 3.16: Kueri SQL terfilter berdasarkan client ID

Penerapan kueri pada Kode 3.16 secara efektif mencegah akses ilegal antar klien. Meskipun seorang klien mengetahui ID faktur milik pihak lain, sistem tidak akan mengembalikan data tersebut karena filter `clientId` tidak akan terpenuhi dalam hasil pencarian basis data.

3.3.8 Reimplementasi Endpoint Get All Payments Berdasarkan Klien

Reimplementasi ini bertujuan untuk memastikan bahwa akses terhadap data pembayaran dikunci secara ketat berdasarkan identitas klien yang terautentikasi melalui token JWT.

A Payment Handler

Pada lapisan ini, fokus utama adalah ekstraksi identitas klien (`userId`) yang telah diinjeksi oleh *middleware* ke dalam konteks lokal Fiber. Potongan kode pada Kode 3.17 menunjukkan bagaimana identitas ini digunakan sebagai parameter utama.

```
// Mengambil userId dari konteks sesi yang terautentikasi
```

```
userId := c.Locals("userId").(string)
payments, err := h.service.GetAllPaymentByClientIdService(userId)
```

Kode 3.17: Ekstraksi identitas klien pada handler

B Payment Service

Lapisan *service* menangani logika transisi data dan validasi hasil. Bagian terpenting adalah penanganan kondisi ketika catatan pembayaran tidak ditemukan dalam basis data, sebagaimana diilustrasikan pada Kode 3.18.

```
// Memastikan pesan error yang informatif saat data kosong
if errors.Is(err, gorm.ErrRecordNotFound) {
    return nil, errors.New("payment tidak ditemukan")
}
```

Kode 3.18: Logika penanganan record tidak ditemukan

C Payment Repository

Melalui operasi *JOIN* antar tabel, sistem memastikan bahwa data pembayaran hanya diambil jika terkait langsung dengan ID klien yang meminta. Mekanisme ini memanfaatkan relasi antara tabel `payments`, `invoices`, dan `clients`, sehingga akses data pembayaran selalu dibatasi oleh kepemilikan klien yang sah. Struktur kueri SQL tersebut dirinci pada Kode 3.19.

```
query := `
    SELECT p.payment_date, p.amount_paid, p.proof_of_transfer
    FROM payments p
    JOIN invoices i ON p.invoice_id = i.invoice_id
    JOIN clients c ON i.client_id = c.client_id
    WHERE c.client_id = ?
    ORDER BY p.payment_date DESC;`

err := p.db.Raw(query, clientId).Scan(&payments).Error
```

Kode 3.19: Kueri JOIN dengan filter keamanan

Atribut data yang diambil dalam kueri tersebut selaras dengan struktur tabel pembayaran yang digunakan oleh sistem. Tabel ini menyimpan informasi inti terkait transaksi pembayaran, termasuk waktu pembayaran, jumlah yang dibayarkan, serta bukti transfer. Struktur lengkap tabel pembayaran ditunjukkan

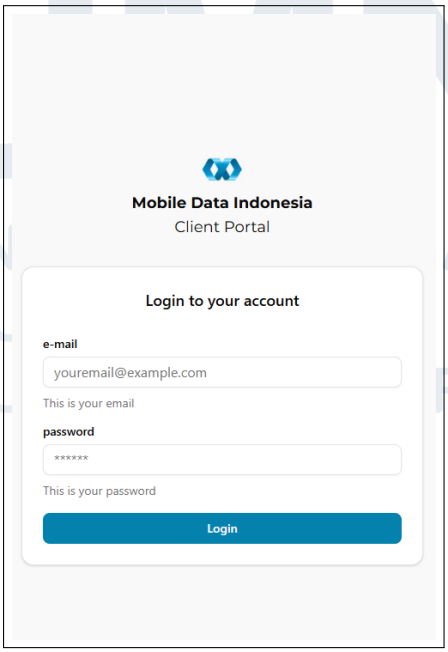
pada Tabel 3.12, yang menjadi dasar dalam proses pengambilan data pada lapisan *repository*.

Tabel 3.12. Struktur table payment

Atribut	Tipe Data
payment_id	String (PK)
payment_date	DateTime
amount_paid	Float
proof_of_transfer	String
voided_at	DateTime
created_at	DateTime
updated_at	DateTime

3.3.9 Penerapan Desain Halaman Login

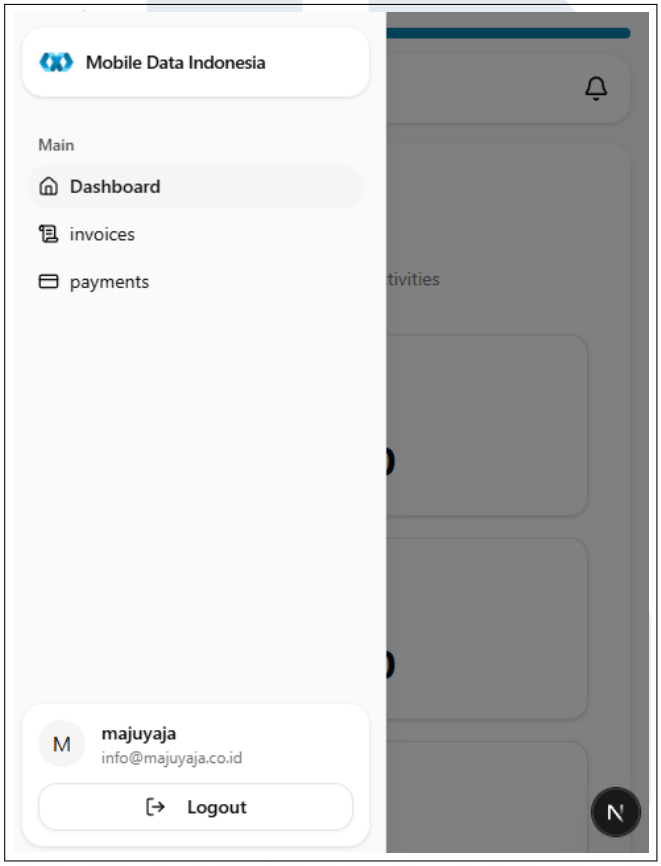
Pada periode minggu ke-5, halaman login dikembangkan, beralih dari fase *prototype* pengujian fungsi menuju penyesuaian penuh dengan desain antarmuka pengguna (UI) yang telah ditetapkan. Implementasi desain baru ini memastikan keselarasan antara fungsi autentikasi yang sudah stabil dan tampilan akhir produk. Hasil akhir dari pembaruan desain halaman login dapat dilihat pada Gambar 3.3.



Gambar 3.3. Tampilan akhir antarmuka halaman login portal klien

3.3.10 Implementasi Side Navigation Bar

Pada minggu ke-6, implementasi side navigation bar dilakukan berdasarkan desain yang telah ditetapkan. Implementasi ini berfungsi untuk memfasilitasi *user* dalam bernavigasi dari satu halaman ke halaman lain. Aspek *responsive design* juga diterapkan guna menjamin kemudahan akses dan navigasi bagi *user* dari berbagai perangkat. Berikut adalah hasil akhir dari implementasi *side navigation bar*.



Gambar 3.4. Tampilan akhir navigation bar

3.3.11 Optimasi Fetching Data dengan Strategi useSWR

Pada minggu ke-6, dilakukan transisi mekanisme pengambilan data dari metode imperatif (*axios*) menjadi deklaratif menggunakan pustaka `useSWR` (*Stale-While-Revalidate*). Strategi ini diterapkan untuk mengoptimalkan performa aplikasi melalui mekanisme *caching* dan sinkronisasi data otomatis di latar belakang.

Penerapan utama `useSWR` difokuskan pada fleksibilitas pengambilan data secara kondisional, di mana permintaan API hanya akan dieksekusi apabila token

otorisasi telah tersedia dalam status aplikasi. Implementasi inti dari logika ini dipaparkan pada Kode 3.20.

```
// Menggunakan SWR untuk caching dan revalidasi otomatis
const { data, error, isLoading } = useSWR<InvoiceResponse>(
    // Conditional fetching: key bernilai null jika token tidak ada
    jwtToken ? `${process.env.NEXT_PUBLIC_API_URL}/api/invoices/get` : null,
    (url: string) => fetcherWithAuth<InvoiceResponse>(url, jwtToken || undefined)
);
```

Kode 3.20: Implementasi fetching Kondisional dengan useSWR

Penggunaan *conditional key* pada Kode 3.20 menjamin efisiensi sumber daya jaringan dengan mencegah pemanggilan *endpoint* yang tidak sah sebelum pengguna terautentikasi. Selain itu, fungsi `fetcherWithAuth` secara otomatis menyuntikkan *bearer token* ke dalam *header* permintaan, sehingga aspek keamanan tetap terjaga tanpa menambah kompleksitas pada komponen antarmuka.

Melalui skema ini, data faktur yang telah diambil akan disimpan dalam *cache* lokal, sehingga saat pengguna berpindah halaman dan kembali lagi, data dapat langsung ditampilkan secara instan tanpa menunggu proses jaringan selesai (*stale data*), sementara validasi data terbaru tetap berjalan di latar belakang.

3.3.12 Implementasi Generate & Download PDF

Fitur pembuatan dokumen PDF dilakukan melalui *API Route* yang berfungsi sebagai perantara (*proxy*) untuk menjaga keamanan token JWT saat melakukan permintaan dokumen ke *backend*.

A Pemrosesan Parameter Permintaan

Implementasi pada sisi *API Route* difokuskan pada ekstraksi data identitas faktur dan kredensial akses yang dikirimkan oleh *frontend*. Inti dari pengambilan data tersebut dapat dilihat pada Kode 3.21.

```
// Ekstraksi data dan token dari body request
const { invoice_id, jwt_token } = await request.json();

// Fetching data ke backend menggunakan Bearer Token
```

```
const apiResponse = await fetch(`${process.env.API_URL}/api/
    invoices/get/detail`, {
    method: "POST",
    headers: { "Authorization": `Bearer ${jwt_token}`, "Content-
    Type": "application/json" },
    body: JSON.stringify({ invoice_id })
});
```

Kode 3.21: Ekstraksi parameter dan otorisasi backend

Penggunaan Kode 3.21 sangat penting karena `jwt_token` digunakan untuk memberikan otorisasi kepada server *backend* agar mengizinkan akses data sensitif. Dengan memproses ini di *server-side*, token tidak terekspos pada URL permintaan di sisi klien.

B Mekanisme Penyimpanan dan Pengiriman PDF

Dokumen PDF tidak disimpan ke dalam *file system* server, melainkan diolah sepenuhnya di dalam memori (*RAM*) dalam bentuk *Buffer*. Hal ini bertujuan untuk mencegah penumpukan berkas sementara dan menjaga kerahasiaan data. Proses pengiriman data biner tersebut ditunjukkan pada Kode 3.22.

```
// Render komponen React menjadi Buffer biner di memori
const pdfBuffer = await renderToBuffer(React.createElement(
    PDFTemplate, { invoiceData }));

// Mengirimkan buffer langsung sebagai respons HTTP
return new NextResponse(pdfBuffer as BodyInit, {
    status: 200,
    headers: {
        "Content-Type": "application/pdf",
        "Content-Disposition": `inline; filename="invoice-${
    invoice_id}.pdf"`,
    },
});
```

Kode 3.22: Konversi ke buffer dan binary streaming

Berdasarkan Kode 3.22, PDF dikirimkan kepada pengguna sebagai aliran data biner (*Binary Stream*). Melalui penetapan *header* `Content-Type: application/pdf`, peramban klien diperintahkan untuk langsung membuka atau mengunduh dokumen tanpa ada jejak berkas fisik yang tertinggal di media

penyimpanan server. Mekanisme *on-the-fly rendering* ini memastikan skalabilitas aplikasi karena server tidak perlu mengelola siklus hidup berkas statis di dalam *disk*.

3.3.13 Implementasi Base64 Sebagai Penyimpanan Gambar

Untuk mendukung arsitektur *stateless*, penyimpanan gambar dilakukan dengan mengonversi berkas biner menjadi format *string* di dalam basis data. Hal ini menghilangkan kebutuhan akan *file system* permanen di server.

A Encoding Sisi Backend

Pada sisi *backend*, fokus utama terletak pada transformasi *buffer* gambar menjadi skema *Data URI* Base64. Inti dari proses konversi tersebut ditunjukkan pada Kode 3.23.

```
// Konversi buffer biner menjadi string Base64 dengan metadata mimetype
const base64String = `data:${req.file.mimetype};base64,${req.file.buffer.toString('base64')}`;
req.body.proof_of_transfer = base64String;
```

Kode 3.23: Transformasi buffer ke data URI base64

B Decoding Sisi Frontend

Di sisi klien, data Base64 direkonstruksi kembali menjadi objek biner (*Blob*) secara temporer di dalam memori peramban. Kode 3.24 menampilkan logika inti pemrosesan data tersebut sebelum ditampilkan kepada pengguna.

```
// Dekode string Base64 menjadi array byte biner
const byteCharacters = atob(proof.split(",")[1]);
const byteArray = new Uint8Array(Array.from(byteCharacters, char => char.charCodeAt(0)));

// Pembuatan URL objek temporer dari Blob
const blobUrl = URL.createObjectURL(new Blob([byteArray], { type: contentType }));
window.open(blobUrl, "_blank");
```

Kode 3.24: Konversi base64 ke blob object

Penerapan Kode 3.23 dan Kode 3.24 memastikan bahwa sistem tidak menyisakan jejak berkas fisik di media penyimpanan server maupun klien. Gambar hanya menempati ruang di basis data sebagai teks, dan diproses di dalam memori (*RAM*) peramban hanya saat dibutuhkan untuk visualisasi. Mekanisme `URL.createObjectURL` memastikan efisiensi karena data biner tersebut tidak perlu diunduh ulang melalui permintaan jaringan tambahan.

3.3.14 Penerapan Email CRON Job

CRON Job merupakan mekanisme penjadwalan standar pada sistem operasi berbasis Unix, yang berfungsi menjalankan perintah atau skrip secara otomatis pada waktu atau interval tertentu [7, 8]. Implementasi *CRON Job* di sini sangat penting untuk mengotomatisasi pengiriman *email reminder* kepada semua klien tanpa memerlukan intervensi manual. *CRON Job* yang dirancang bertujuan untuk mengirimkan *email* secara berkala pada interval yang telah ditentukan. Berikut adalah implementasi kode untuk mengaktifkan penjadwalan tersebut.

```
1    cronJob := cron.New()
2    cronJob.AddFunc("0 7 * * *", func() { jobs.EmailCron(config.DB
     ) })
3    cronJob.Start()
```

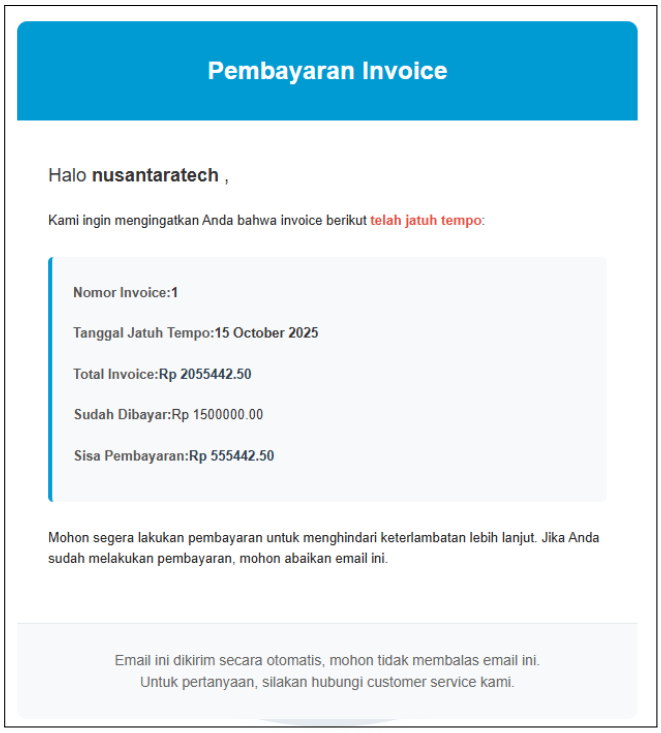
Kode 3.25: Inisialisasi dan penjadwalan tugas email

Kode 3.25 menunjukkan inisialisasi *scheduler* CRON baru (`cronJob := cron.New()`). Fungsi `AddFunc` digunakan untuk mendaftarkan tugas, di mana `"0 7 * * *"` merupakan sintaks CRON yang menjadwalkan eksekusi tugas pada pukul 07:00 pagi setiap hari. Tugas yang dijalankan adalah `jobs.EmailCron`, yang bertanggung jawab mengirim *email reminder* menggunakan koneksi *database* yang disediakan (`config.DB`). Akhirnya, `cronJob.Start()` mengaktifkan *scheduler* tersebut.

3.3.15 Email Notification

Email notification diimplementasikan untuk memberikan *reminder* pembayaran *invoice* secara langsung kepada *client* tanpa memerlukan intervensi manual dari pihak perusahaan. Setiap notifikasi mencakup informasi yang jelas mengenai batas waktu pembayaran (*deadline*) yang tersisa. Otomatisasi ini memastikan *client* menerima informasi tepat waktu, sehingga meningkatkan

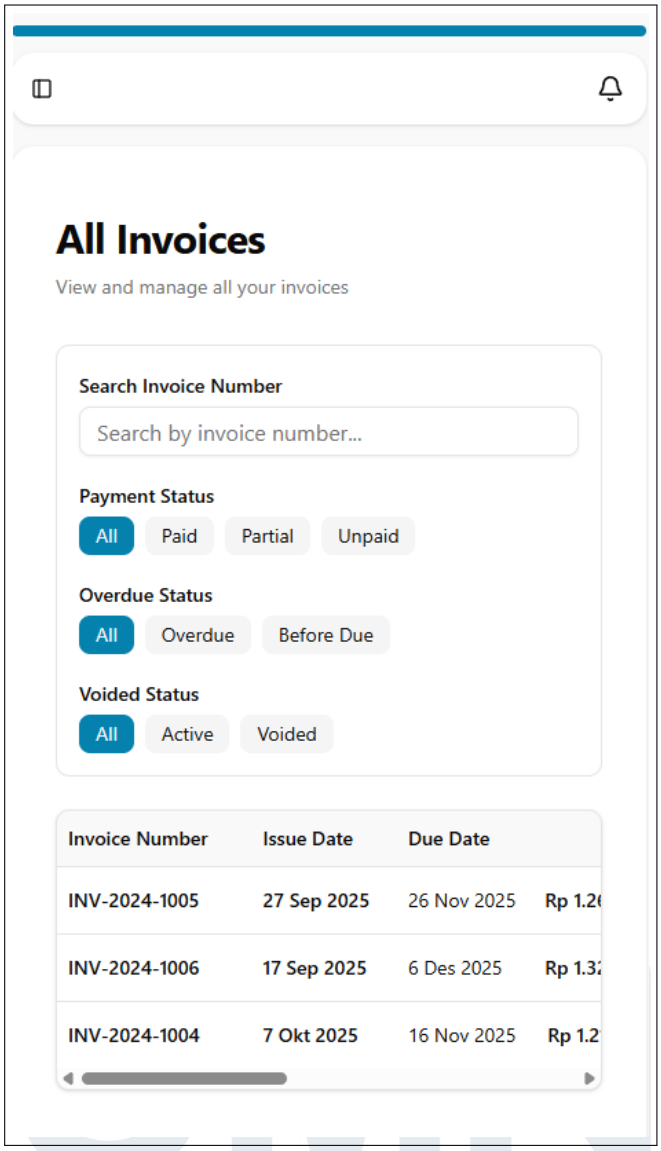
efisiensi proses penagihan. Tampilan contoh *email reminder* yang dikirimkan kepada *client* dapat dilihat pada Gambar 3.5.



Gambar 3.5. Tampilan contoh email notifikasi pembayaran invoice

3.3.16 Implementasi Halaman Invoice

Pada minggu ke-10 hingga ke-11, Halaman Invoice telah berhasil diimplementasikan dan diselesaikan sesuai dengan desain yang telah disetujui sebelumnya. Halaman ini berfungsi sebagai pusat bagi client untuk melihat dan mengelola semua faktur mereka. Tampilan hasil akhir dari Halaman Invoice dapat dilihat pada Gambar 3.6.



Gambar 3.6. Tampilan akhir halaman invoice

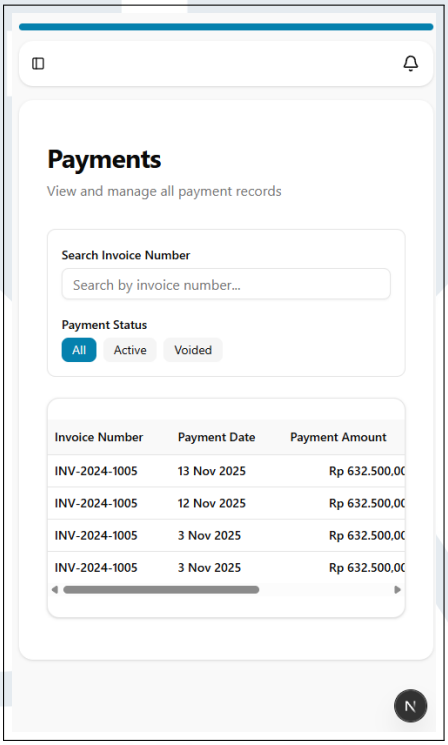
Halaman Invoice ini dilengkapi dengan fitur filter dan tampilan tabel data. Berikut rincian fitur filter dan deskripsi kolom tabel.

3.3.17 Implementasi Halaman Payment

Pada minggu ke-10 hingga ke-11, implementasi *Halaman Payment* berhasil diselesaikan. Halaman ini berfungsi sebagai pusat bagi *client* untuk meninjau dan melacak semua riwayat pembayaran yang telah dilakukan untuk setiap *invoice*. Fitur yang disediakan mencakup *filter* dan *search* sederhana, serta kapabilitas *sorting* pada kolom tabel. Tampilan halaman ini dapat dilihat pada Gambar 3.7.

Tabel 3.13. Rincian fitur filter halaman invoice

Fitur Filter	**Fungsi**
Search Invoice Number	Berfungsi untuk mencari *invoice* secara spesifik berdasarkan nomor faktur (`Invoice Number`).
Payment Status	Memungkinkan *user* memfilter *invoice* berdasarkan status pembayarannya (misalnya: *Paid*, *Unpaid*).
Overdue Status	Memungkinkan *user* memfilter *invoice* berdasarkan status jatuh temponya (*Overdue* atau *Before Due*).
Voided Status	Memungkinkan *user* memfilter *invoice* berdasarkan status validitasnya (*Active* atau *Voided*).



Gambar 3.7. Tampilan akhir halaman payment

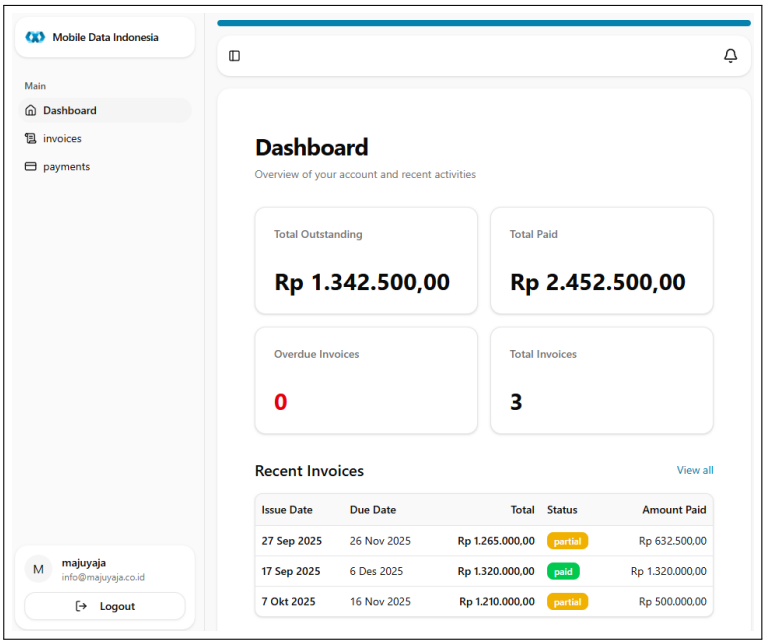
Berikut adalah rincian fitur *filter* dan deskripsi kolom tabel yang terdapat pada *Halaman Payment*.

Tabel 3.14. Rincian fitur filter halaman payment

Fitur Filter	Fungsi
Search	Berfungsi untuk mencari pembayaran berdasarkan invoice number secara spesifik.
Filter	Memungkinkan *user* memfilter data pembayaran berdasarkan status pembayaran.

3.3.18 Implementasi Halaman Dashboard

Implementasi *Halaman Dashboard* juga dilaksanakan pada minggu ke-10 hingga ke-11. *Dashboard* ini berfungsi untuk menampilkan ringkasan data penting (*key metrics*) secara menyeluruh, sehingga memberikan gambaran cepat mengenai status finansial perusahaan kepada *client*. Informasi utama yang disajikan meliputi Total Tagihan, Total Nominal yang Sudah Dibayar, jumlah *invoice* yang sudah melewati *deadline* (*Overdue*), dan Total Keseluruhan Invoice. Selain itu, disajikan pula daftar *Recent Invoice* yang menampilkan faktur yang baru saja diterbitkan, sehingga memudahkan *client* dalam pelacakan aktivitas terbaru. Tampilan akhir dari *Halaman Dashboard* dapat dilihat pada Gambar 3.8.



Gambar 3.8. Tampilan akhir halaman dashboard

3.3.19 Implementasi Rate Limiter

Pada minggu ke-12, *rate limiter* diimplementasikan sebagai lapisan keamanan yang bertujuan untuk mengendalikan frekuensi permintaan (*request*) dari setiap klien. Hal ini penting untuk menjaga penggunaan sumber daya *server* tetap stabil dan mencegah serangan *Denial of Service* (DoS) atau penyalahgunaan API. Implementasi *rate limiter* ini menggunakan *middleware* `limiter` yang disediakan oleh *framework* Go Fiber.

```
app.Use(limiter.New(limiter.Config{
  Max        : 10,
  Expiration: 30 * time.Second,
```

Kode 3.26: Konfigurasi dasar dan batas akses

Kode 3.26 menunjukkan inisialisasi *middleware* `limiter` dan penerapannya pada seluruh aplikasi (`app.Use`). Konfigurasi dasar menetapkan batas (`Max`) maksimum 10 permintaan dalam durasi (`Expiration`) 30 detik. Ini berarti setiap alamat IP hanya diizinkan mengirim 10 *request* dalam periode 30 detik.

```
  KeyGenerator: func(c *fiber.Ctx) string {
    return c.IP()
  },
  LimitReached: func(c *fiber.Ctx) error {
    return utils.Error(c, fiber.StatusTooManyRequests, "Too Many
  Request", "Please try again later")
  },
}))
```

Kode 3.27: Penentuan kunci (key) dan penanganan batas tercapai

Pada kode 3.27, `KeyGenerator` diatur untuk mengidentifikasi klien berdasarkan alamat IP mereka (`c.IP()`), sehingga setiap IP memiliki batas permintaannya sendiri. Fungsi `LimitReached` mendefinisikan respons yang akan dikirimkan ketika batas permintaan terlampaui. Respons yang dikirim adalah *`429 Too Many Requests`* dengan pesan *error* kustom yang menyarankan klien untuk mencoba lagi nanti.

3.3.20 Implementasi Protected Routes

Perlindungan rute (*protected routes*) pada sisi *frontend* diterapkan untuk memastikan akses ke halaman sensitif seperti *dashboard*, *invoices*, dan *payments* terbatas hanya bagi klien yang memiliki kredensial sah. Mekanisme ini

menggunakan *middleware* yang bekerja di tingkat *edge runtime* untuk mencegat permintaan sebelum mencapai komponen halaman.

A Logika Navigasi dan Redireksi Middleware

Middleware memeriksa keberadaan *session token* di dalam *cookies* klien pada setiap transisi halaman. Fokus utama dari logika ini adalah mencegah pengguna yang sudah *login* untuk kembali ke halaman login, serta memblokir akses anonim ke jalur terproteksi. Implementasi ringkas dari logika tersebut dapat dilihat pada Kode 3.28.

```
export function middleware(request: NextRequest) {
    const sessionToken = request.cookies.get("access_token")?.
    value;
    const isLoginPage = request.nextUrl.pathname === "/login";

    // Mencegah akses ke halaman login jika sudah terautentikasi
    if (isLoginPage && sessionToken) {
        return NextResponse.redirect(new URL("/", request.nextUrl.
    origin));
    }

    // Proteksi rute: Arahkan ke login jika tidak ada token
    if (!isLoginPage && !sessionToken) {
        return NextResponse.redirect(new URL("/login", request.
    nextUrl.origin));
    }

    return NextResponse.next();
}
```

Kode 3.28: Logika validasi sesi dan redireksi

Penerapan Kode 3.28 memastikan bahwa aliran navigasi pengguna selalu sinkron dengan status autentikasi mereka. Dengan melakukan pemeriksaan di sisi server (*middleware*), sistem dapat menghindari kebocoran data visual yang sering terjadi jika proteksi hanya dilakukan di sisi klien (*client-side rendering*).

B Konfigurasi Jalur Terproteksi

Untuk mengoptimalkan performa, *middleware* hanya dijalankan pada jalur-jalur tertentu yang didefinisikan dalam objek konfigurasi. Hal ini mencegah

eksekusi kode pada berkas statis atau aset publik. Daftar jalur yang dilindungi oleh sistem ini dirinci pada Kode 3.29.

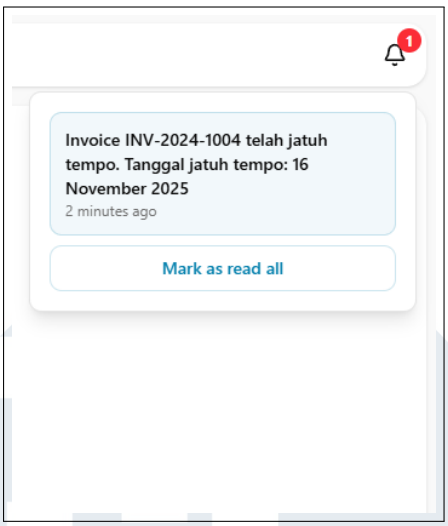
```
export const config = {
    matcher: [
        "/",
        "/login",
        "/invoices/:path*",
        "/payments/:path*",
    ],
};
```

Kode 3.29: Konfigurasi matcher middleware

Melalui penggunaan *matcher* pada Kode 3.29, sistem secara otomatis memetakan rute mana saja yang memerlukan validasi token. Kombinasi antara logika pada Kode 3.28 dan konfigurasi rute ini menciptakan sistem keamanan *frontend* yang tangguh dan efisien dalam pengelolaan sesi pengguna.

3.3.21 Pengembangan Sistem Notifikasi

Sistem notifikasi dikembangkan untuk melengkapi fungsi *CRON Job* yang telah dibuat sebelumnya. Setelah *CRON Job* mengirimkan *email reminder* pembayaran *invoice*, sistem secara bersamaan akan membuat dan menyimpan notifikasi baru yang ditujukan kepada *client* yang bersangkutan. Notifikasi ini berfungsi untuk memberi tahu *user* mengenai invoice yang mendekati batas waktu pembayaran (*deadline*) atau yang sudah melewati jatuh tempo. Tampilan antarmuka notifikasi dapat dilihat pada Gambar 3.9.



Gambar 3.9. Tampilan modal notifikasi

Pada antarmuka, setiap notifikasi baru yang belum dibaca akan memicu munculnya indikator mengambang (*floating notification*) pada ikon bel, yang menunjukkan jumlah notifikasi yang belum dibaca. Ketika ikon tersebut diklik, akan muncul *popover modal* yang menampilkan daftar (*list*) notifikasi. *User* dapat menggunakan tombol *Mark As Read* untuk menandai semua notifikasi sebagai sudah dibaca, yang secara otomatis akan menghilangkan indikator mengambang pada ikon notifikasi.

3.3.22 Penambahan Seeding Data

Pada minggu terakhir pelaksanaan, dilakukan penambahan volume *seeding data* secara signifikan untuk menciptakan skenario simulasi yang mendekati kondisi nyata (*real case*). Fokus utama adalah menghasilkan dataset *invoice* dengan status pembayaran yang bervariasi (*unpaid*, *partial*, *paid*) guna menguji akurasi filter dan kalkulasi saldo pada aplikasi.

A Invoice Seeder

Invoice Seeder dirancang untuk menghasilkan tiga faktur per klien dengan skenario status yang berbeda. Bagian krusial dari implementasi ini adalah penggunaan blok `switch` untuk menentukan *logic* nominal pembayaran dan pembungkusan kueri ke dalam *database transaction* untuk menjamin integritas data antara tabel *invoice* dan detailnya. Logika inti tersebut ditunjukkan pada Kode 3.30.

```
// Loop untuk menghasilkan 3 status berbeda per klien
for i := 0; i < 3; i++ {
    var paymentStatus string
    var amountPaid float64

    switch i {
    case 0: paymentStatus, amountPaid = "unpaid", 0
    case 1: paymentStatus, amountPaid = "partial", total / 2
    case 2: paymentStatus, amountPaid = "paid", total
    }

    // Menjamin integritas data menggunakan Transaksi GORM
    db.Transaction(func(tx *gorm.DB) error {
        // ... Insert Invoice ...
        // Menggunakan RETURNING invoice_id untuk relasi detail
        tx.Raw(invoiceQuery, ...).Scan(&invoiceID)

        // Insert Invoice Detail berdasarkan ID yang baru dibuat
        return tx.Exec(invoiceDetailQuery, invoiceID, ...).Error
    })
}
```

Kode 3.30: Logika penentuan status dan transaksi database

Melalui penggunaan Kode 3.30, sistem dapat mensimulasikan berbagai kondisi keuangan klien dalam satu kali eksekusi, sehingga pengujian fitur *reporting* menjadi lebih komprehensif.

B Payment Seeder

Payment Seeder melengkapi dataset dengan membuat catatan pembayaran hanya untuk faktur yang tidak berstatus *unpaid*. Langkah selektif yang diambil dalam kueri ini adalah melakukan filter awal pada status *invoice* sebelum melakukan proses penyisipan data ke tabel `payments`. Implementasi kueri dan kalkulasi sisa tagihan dipaparkan pada Kode 3.31.

```
// Hanya mengambil invoice yang memiliki riwayat pembayaran (
    partial/paid)
db.Raw('SELECT invoice_id, total, amount_paid FROM invoices
        WHERE payment_status IN ('partial', 'paid') LIMIT 10').
    Scan(&invoices)

for _, inv := range invoices {
```

```
    remainingAmount := inv.Total - inv.AmountPaid
    if remainingAmount > 0 {
        // Membuat record pembayaran untuk sisa tagihan
        paymentQuery := `INSERT INTO payments (payment_id,
    amount_paid, invoice_id, ...)
                        VALUES (gen_random_uuid(), ?, ?, ...)`
        db.Exec(paymentQuery, remainingAmount, inv.InvoiceID, ...)
    }
}
```

Kode 3.31: Filtering invoice dan pembuatan data pembayaran

Berdasarkan Kode 3.31, data pembayaran yang dihasilkan selalu sinkron dengan sisa saldo pada faktur terkait. Dengan memisahkan tugas *seeder* menjadi dua tahap ini, pengembang dapat memastikan bahwa relasi antar-tabel diuji secara vertikal, mulai dari entitas klien hingga ke catatan pembayaran spesifik.

3.4 Kendala dan Solusi yang Ditemukan

Selama proses migrasi dan pengembangan *backend* pada proyek baru, terdapat beberapa kendala teknis dan operasional yang terkait dengan kompatibilitas sistem lama dan adaptasi teknologi baru.

1. Adanya *learning curve* untuk bahasa pemrograman Go, khususnya dalam konsep penggunaan *struct* dan *receiver*, menyebabkan proses pengembangan berjalan kurang efisien. Hal ini disebabkan oleh perbedaan paradigma pemrograman (seperti *OOP*) dengan bahasa yang telah dikuasai sebelumnya.
2. *ORM* dari proyek *Service Internal Invoice* tidak dapat digunakan kembali (*reused*) untuk pengembangan aplikasi *Client Portal* karena ketidakcocokan teknologi dan struktur data antara kedua *codebase* yang independen.
3. Mekanisme *upload* gambar pada aplikasi *Service Invoice Internal* masih menyimpan file di folder lokal (*uploads*), sehingga tidak dapat diakses dan ditampilkan oleh aplikasi *Service Client Portal*. Hal ini mengharuskan perubahan pada *codebase* untuk memindahkan lokasi penyimpanan ke tempat yang dapat diakses bersama.

Pada setiap kendala yang disebutkan, berikut solusi yang diimplementasikan:

1. Pembelajaran intensif secara mandiri melalui dokumentasi dan tutorial Golang untuk mempercepat adaptasi *syntax* pemrograman.

2. Penerapan *Raw Query* SQL secara langsung untuk menangani operasi *database*, memastikan kompatibilitas dengan skema yang ada dan menghindari ketergantungan pada *ORM* lama.

3. Modifikasi kode untuk mengganti mekanisme penyimpanan lokal dengan integrasi layanan penyimpanan terpusat agar file dapat diakses bersama oleh berbagai layanan aplikasi.