BAB 3
PELAKSANAAN KERJA MAGANG

3.1 Kedudukan dan Koordinasi

Posisi yang diberikan selama menjalani kegiatan magang di PT Mobile Data Indonesia adalah sebagai *Staff IT Developer*. Posisi ini kedudukannya di bawah *Head IT* dan merupakan bagian dari Tim IT dari PT Mobile Data Indonesia. Adapun posisi dari *Head IT* diemban oleh Bapak Iwan Hasim. Beliau yang bertugas dalam membimbing dan mengawasi pekerjaan selama magang. Tugas utama yang diberikan oleh perusahaan awalnya adalah membuat aplikasi *mobile* untuk inventaris kantor, namun, kemudian dirubah untuk membuat sistem *invoicing* berbasis *website* sesuai dengan kebutuhan perusahaan. Lalu untuk koordinasi dilakukan dengan berbicara langsung secara tatap muka di kantor dan menggunakan aplikasi *messaging* berupa WhatsApp ketika tidak bisa dilakukan secara langsung atau tatap muka. Progres dari pembuatan *website* ditanyakan setiap harinya dan pengujian dilakukan saat satu fitur selesai hingga seluruh *project* selesai dibuat.

3.2 Tugas yang Dilakukan

Selama pelaksanaan magang di PT Mobile Data Indonesia, pekerjaan yang dilakukan meliputi perancangan *website*, pengembangan *back-end*, integrasi dengan *front-end*, serta pembuatan fitur pendukung lainnya. Berikut tugas-tugas yang dilakukan.

1. Membantu dalam memilih teknologi yang digunakan dalam membangun *website*.
2. Merancang tabel untuk *payment* dan *log*.
3. Membuat fitur *payment* untuk *back-end*.
4. Membuat fitur untuk *logging* dari setiap aktifitas di *back-end*.
5. Melakukan integrasi antara *back-end* dan *front-end* pada fitur *client* di *Client Page*.
6. Melakukan integrasi antara *back-end* dan *front-end* pada fitur *invoice* di *Invoice Page*.

7. Melakukan integrasi antara *back-end* dan *front-end* pada fitur *payment* di *Payment Page*.

8. Melakukan integrasi antara *back-end* dan *front-end* pada fitur *report* di *Report Page*.

3.3 Uraian Pelaksanaan Magang

Berikut merupakan tabel 3.1 yang berisi uraian dari pelaksanaan magang yang dilakukan di PT Mobile Data Indonesia.

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

Minggu Ke -	Pekerjaan yang dilakukan
1	Perkenalan perusahaan dan pembekalan mengenai *job desc* yang akan dilakukan.
2	Belajar *framework* Flutter dari dokumentasi dan internet.
3	Diskusi mengenai perancangan aplikasi untuk manajemen inventaris kantor
4	Belajar membuat aplikasi *dummy* berupa *to-do list* dengan Flutter sembari menunggu hasil desain.
5	Belajar membuat aplikasi *dummy* berupa *Weather App* untuk melihat cuaca terkini dengan menggunakan *Application Programming Interface* (API) di Flutter sembari menunggu hasil desain.
6	Belajar membuat aplikasi *dummy* berupa *dashboard* sederhana menggunakan Flutter sembari menunggu hasil desain.
7	Belajar lebih lanjut bagaimana melakukan integrasi API dengan Flutter sembari menunggu desain.
8	Belajar penggunaan *state management* (*Provider*) di Flutter sembari menunggu hasil desain.
9	Diskusi mengenai perubahan *project* ke pembuatan *website* untuk sistem *invoicing* perusahaan dan riset awal untuk *tech stack* yang akan digunakan.
	Lanjut pada halaman berikutnya.

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

Minggu Ke -	Pekerjaan yang dilakukan
10	Belajar mengenai RESTful API dan melanjutkan riset untuk menentukan *framework* dan *library* yang akan digunakan dalam membangun *website invoicing.*
11	Rapat mengenai pembuatan *website* yang baru dan mengumpulkan data yang dibutuhkan dengan cara melakukan *user requirement gathering* kemudian membuat *schema database* dan inisialisasi *project.*
12	Pembuatan fitur *payment* untuk *back-end*, *testing* dan *bug fixing*.
13	Pembuatan fitur untuk dapat melakukan *log* di *back-end* dan *testing*.
14	Melanjutkan implementasi fitur *log* di modul *controller* yang tersisa.
15	Mengimplementasikan Swagger API untuk dokumentasi fitur modul *payment*. Lalu, *review* dan *testing* seluruh fitur yang sudah dibuat di *back-end*.
16	Melakukan integrasi dari *front-end* dan *back-end* pada *Client Page* dan *bug fixing* selama proses integrasi.
17	Melakukan integrasi dari *front-end* dan *back-end* pada *Invoice Page* dan *bug fixing* selama proses integrasi.
18	Melakukan integrasi dari *front-end* dan *back-end* pada *Payment Page* dan *bug fixing* selama proses integrasi.
19	Melakukan integrasi dari *front-end* dan *back-end* pada *Report Page* dan *bug fixing* selama proses integrasi. Lalu *review* dan *testing project* oleh internal Tim IT.
20	Melakukan pengetesan oleh *end user* dari bagian keuangan dan manajemen.

3.4 Perancangan Sistem *Website*

Sebelum tahap implementasi dilakukan, perancangan sistem perlu dilakukan terlebih dahulu untuk memastikan bahwa sistem yang dikembangkan sesuai dengan kebutuhan pengguna. Perancangan ini mencakup pemilihan metode pengembangan

sistem, *user requirement gathering*, identifikasi aktor dan fitur-fitur utama melalui diagram *use case*, pemetaan alur proses melalui *flowchart* dan penyusunan struktur *database* yang akan digunakan dalam pengelolaan data *website invoicing*. Semua elemen perancangan ini menjadi landasan penting dalam pengembangan *website* sistem *invoicing* di PT Mobile Data Indonesia.

3.4.1 Metode Pengembangan Sistem

Sistem invoicing ini dikembangkan dengan menggunakan metode *Software Development Life Cycle* (SDLC). SDLC merupakan kerangka kerja yang terdiri dari beberapa tahapan yaitu analisis kebutuhan, perancangan, implementasi, dan pengetesan [7]. Metode ini digunakan agar proses pengembangan sistem bisa dilakukan secara sistematis dan terkontrol [8].

A Model *Waterfall*

Pada *project* ini, Model *Waterfall* digunakan sebagai acuan model pengembangan. Model *Waterfall* merupakan salah satu model SDLC yang paling awal dan sering digunakan dalam pengembangan perangkat lunak. Karakteristik utama dari Model *Waterfall* adalah urutan fase yang harus diselesaikan terlebih dahulu sebelum berpindah ke fase selanjutnya [7]. Berikut tahapan pengembangan dari Model *Waterfall*:

1. ***Requirements Definition***
   Tahap pertama dalam Model *Waterfall* adalah membuat dan mengumpulkan definisi kebutuhan sistem berdasarkan proses bisnis dan kebutuhan pengguna. Informasi yang didapat menjadi dasar dalam menentukan fitur dan ruang lingkup sistem yang akan dibuat.

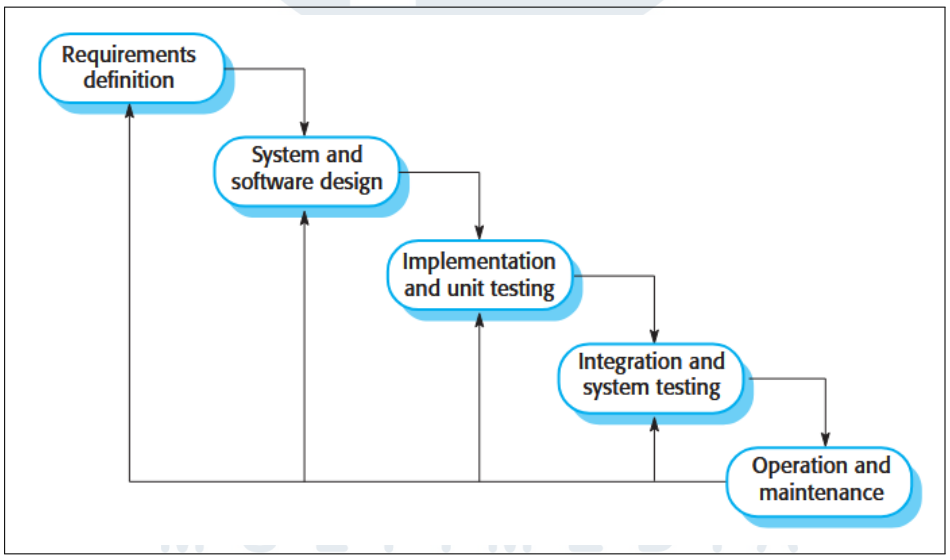
2. ***System and Software Design***
   Setelah kebutuhan didokumentasikan dengan jelas, tahap kedua adalah merancang arsitektur sistem dan desain perangkat lunak. Perancangan meliputi alur proses sistem, struktur data yang digunakan, dan aktor yang menggunakan sistem ini.

3. ***Implementation and Unit Testing***
   Tahap ketiga melakukan implementasi desain menjadi kode sesuai dengan spesifikasi desain dan teknologi yang telah disepakati. Kemudian, setiap

komponen diuji secara terpisah untuk memastikan fungsi berjalan dengan benar sebelum digabungkan dengan komponen lainnya.

4. ***Integration and System Testing***
   Setelah setiap *unit* atau komponen diuji secara terpisah, tahap keempat adalah mengintegrasikan semua komponen sistem menjadi satu kesatuan yang utuh dan melakukan pengujian sistem secara menyeluruh. Pengujian integrasi bertujuan untuk memastikan bahwa semua modul bekerja dengan baik ketika digabungkan dan dapat berkomunikasi dengan benar.

5. ***Operation and Maintenance***
   Terakhir, tahap kelima adalah operasi dan pemeliharaan sistem. Pada tahap ini, sistem yang telah selesai mulai digunakan oleh pengguna. Selama proses penggunaan, pemantauan dan perawatan sistem dilakukan untuk memastikan sistem tetap berjalan stabil serta dilakukan perbaikan apabila ditemukan kendala selama penggunaan.



Gambar 3.1. *SDLC - Model Waterfall*
Sumber: [7]

3.4.2 *User Requirement Gathering*

User requirement gathering merupakan proses analisis yang dilakukan untuk mengidentifikasi kebutuhan pengguna terhadap sistem yang akan

dikembangkan. Melalui proses ini, sistem dapat dirancang sesuai dengan proses bisnis yang berjalan. Kebutuhan pengguna diperoleh melalui diskusi dan wawancara dengan pihak terlibat yang dalam penggunaan sistem *invoicing* berbasis *web* di PT Mobile Data Indonesia.

A *Role Finance*

Role Finance merupakan pihak yang bertanggung jawab terhadap pencatatan, pengelolaan, dan pemantauan transaksi keuangan perusahaan. Oleh karena itu, kebutuhan sistem dari *role Finance* bersifat operasional dan berkaitan langsung dengan pengelolaan data invoice serta pembayaran. Berikut merupakan daftar kebutuhan dari *role Finance*:

1. Pengguna dapat membuat dan mengelola data *invoice*.
2. Pengguna dapat merubah status *invoice* menjadi valid atau tidak valid.
3. Pengguna dapat mencatat dan mengelola data pembayaran yang masuk berdasarkan *invoice* terkait.
4. Pengguna dapat mengunggah dan melihat bukti pembayaran.
5. Pengguna dapat membuat dan mengelola data klien perusahaan.
6. Pengguna dapat melihat laporan pembayaran dan *invoice* secara keseluruhan.

B *Role Management*

Role Management digunakan oleh pihak manajemen perusahaan yang berperan dalam pengambilan keputusan berdasarkan data keuangan. *Role* ini tidak terlibat langsung dalam pengolahan data, sehingga kebutuhan sistem lebih berfokus pada pemantauan dan pelaporan informasi. Berikut merupakan daftar kebutuhan dari *role Management*:

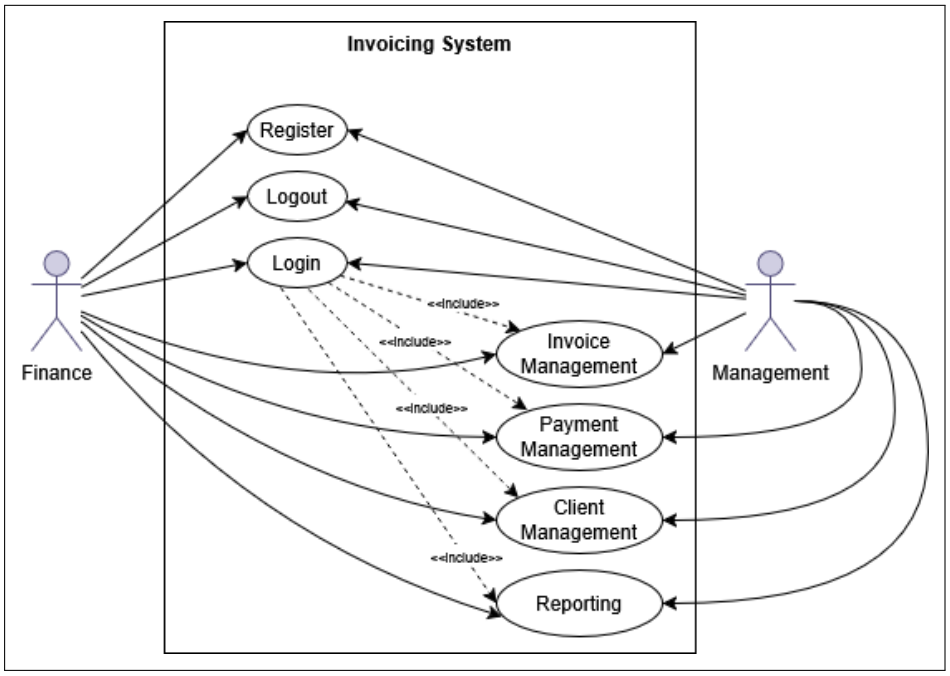
1. Pengguna dapat melihat ringkasan data *invoice* dan pembayaran.
2. Pengguna dapat mengakses laporan keuangan secara keseluruhan.
3. Pengguna dapat memantau status pembayaran klien.
4. Pengguna hanya memiliki akses untuk melihat data tanpa dapat mengubah atau menambahkan data.

3.4.3 *Use Case* Diagram

Use Case diagram merupakan jenis diagram yang menghubungkan antara aktor dan sistem yang sedang dikembangkan. *Use Case* diagram digunakan karena dapat memudahkan pengembang dalam menggambarkan *role* apa saja yang bisa mengakses sistem dan berkoordinasi dengan tim saat pengembangan. Contohnya pada *project* ini, terdapat dua aktor yaitu ”*Finance*” dan ”*Management*.”

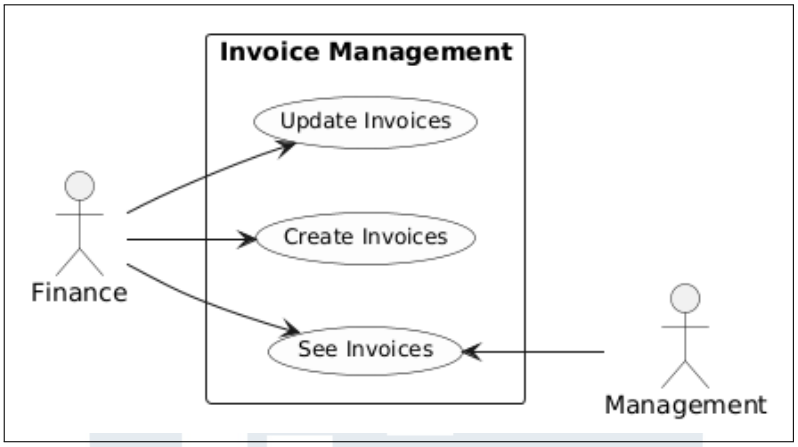
Kedua *role* ini memiliki hak akses yang berbeda. Hal ini dilakukan dengan alasan agar terdapat pembagian tugas yang jelas dan untuk meminimalkan risiko kesalahan atau penyalahgunaan terhadap sistem. Baik *role Finance* dan *Management* memiliki hak akses untuk autentikasi seperti *Register*, *Login*, dan *Logout*. Selain itu, mereka juga memiliki hak akses dalam melakukan manajemen terhadap *Invoice*, *Payment*, *Client*, dan *Reporting*. Namun, untuk *Management* hanya bisa melihat data yang sudah ada saja dan tidak bisa membuat atau melakukan *update*.

Berikut merupakan gambar 3.2 yang menjelaskan *Overview* dari keseluruhan sistem *invoicing*. Untuk modul *Invoicing Management*, *Payment Management*, *Client Management*, dan *Reporting* akan dijelaskan dengan lebih detail di sub-bab selanjutnya.



Gambar 3.2. *Use Case Diagram - Invoicing System Overview*

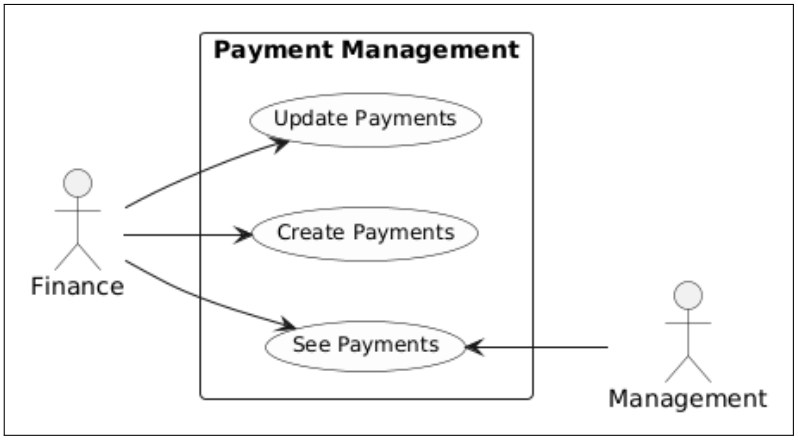
A *Invoice Management*



Gambar 3.3. *Use Case Diagram - Invoice Management*

Pada modul ini, *Invoice Management* bertugas sebagai pusat pengelolaan seluruh aktivitas yang berhubungan dengan *invoice*. Pengguna dapat membuat *invoice* baru, melakukan *update* terhadap *invoice* yang sudah dibuat, dan dapat menampilkan daftar *invoice* pada *website*. *Role finance* mendapatkan akses penuh dalam seluruh pengelolaan yang berhubungan dengan *invoice*, sementara *management* terbatas hanya bisa melihat data tanpa bisa membuat atau merubah data.

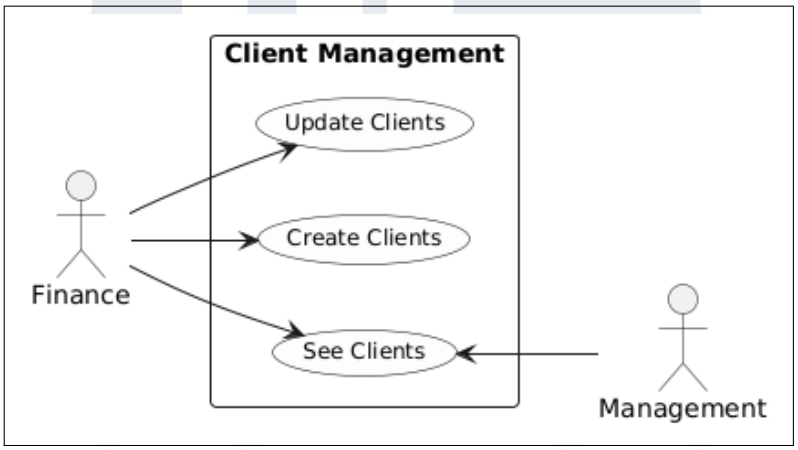
B *Payment Management*



Gambar 3.4. *Use Case Diagram - Payment Management*

Selanjutnya, modul *Payment Management* dirancang untuk menangani semua proses yang berkaitan dengan pembayaran, mulai dari pencatatan pembayaran yang masuk dan *update* data *payment* yang sudah ada di *database*. Pengguna juga bisa melihat data dari *payment*. Sama seperti modul *invoice*, *finance* memiliki akses penuh dalam pengelolaan data *payment*. Sebaliknya, *management* hanya dapat mengakses informasi berupa tampilan saja di *website*. Modul ini ditugaskan pada penulis dari mulai perancangan sampai implementasi.

C *Client Management*



Gambar 3.5. *Use Case Diagram - Client Management*

Kemudian pada modul *Client Management*, fokus utamanya adalah dalam melakukan pengelolaan data klien yang menggunakan jasa dari PT Mobile Data Indonesia. Aktivitas yang bisa dilakukan di antaranya adalah penambahan klien baru, perubahan data klien yang sudah ada di *database*, serta menampilkan seluruh informasi yang berhubungan dengan klien. *User* dengan *role finance* memiliki hak akses untuk melakukan seluruh aktivitas pengelolaan yang berhubungan dengan klien. Sementara itu, *role management* hanya diberi hak akses dalam melihat informasi klien saja.

D Reporting

Gambar 3.6. *Use Case Diagram - Reporting*

Terakhir, modul *Reporting* merupakan modul yang akan menampilkan *invoice* apa saja yang sudah dibuat dan nominal pembayaran yang belum dan sudah dilakukan oleh klien. Penghitungan nominal pembayaran dari *reporting* didapat dari *invoice* dan *payment*. Oleh sebab itu, aktivitas yang tersedia pada modul *reporting* hanya menampilkan data saja.

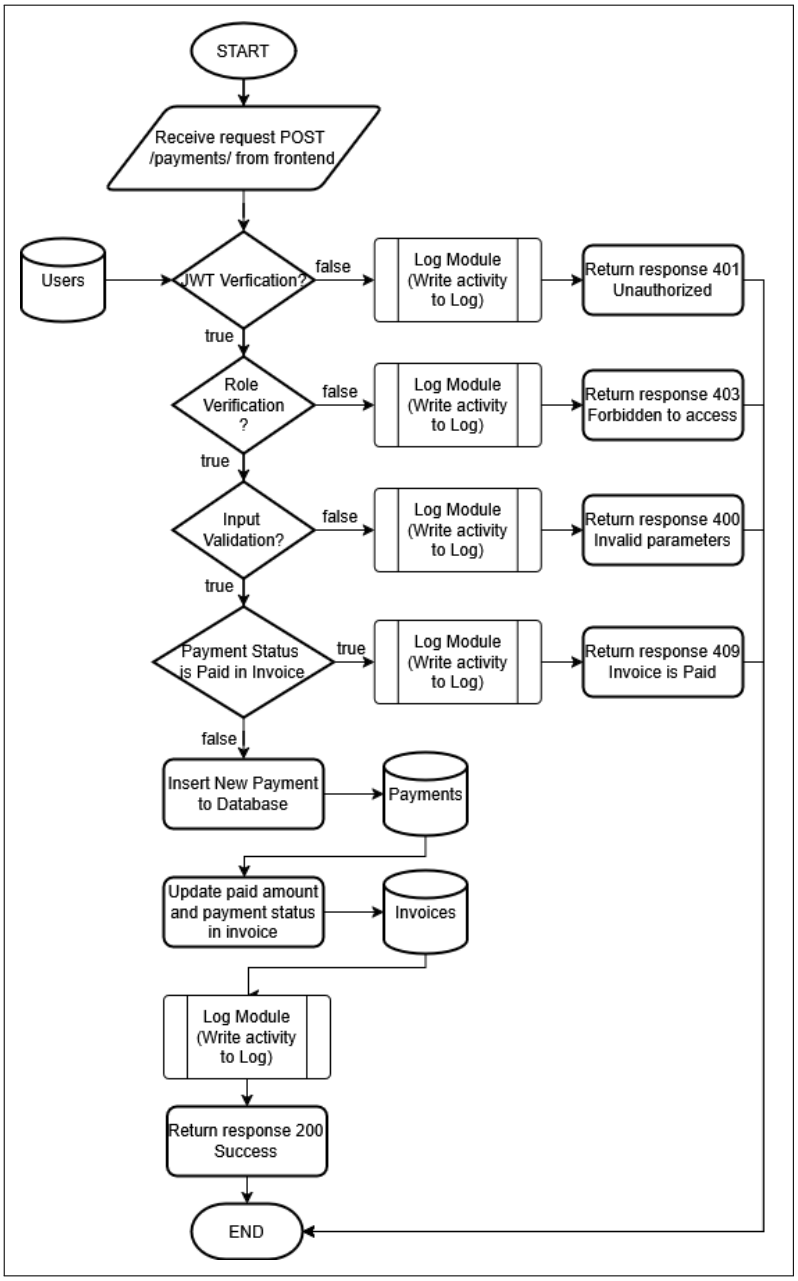
3.4.4 *Flowchart*

Flowchart digunakan untuk menggambarkan alur kerja sebuah sistem. Mulai dari menerima *input* dari pengguna hingga menghasilkan *output* tertentu. Pada sistem *invoicing* yang sedang dibuat, *flowchart* dirancang untuk setiap fungsi yang berkaitan dengan fitur pembayaran dan pencatatan aktivitas (*log*). Fitur-fitur yang dibuat antara lain pembuatan *payment*, modifikasi *payment*, mengambil informasi seluruh atau salah satu yang berkaitan dengan *payment*, melihat bukti transfer, dan merubah status berlaku atau tidaknya sebuah pembayaran. Selain itu ada juga *flowchart* untuk menggambarkan alur bagaimana sebuah aktivitas di simpan di database.

A *Create Payment*

Flowchart ini menggambarkan alur pembuatan data pembayaran yang baru dibuat oleh pengguna. Proses dimulai ketika *server* mendapatkan *request* dengan *method POST* dan *endpoint* berupa */payments/* dari *front-end*. Kemudian, *back-end* melakukan verifikasi akun, *role*, dan *input* yang diberikan oleh pengguna. Selanjutnya, sistem akan mengecek apakah *invoice* yang ingin dibayar sebelumnya sudah lunas atau belum, jika sudah maka *server* akan memberi respons bahwa

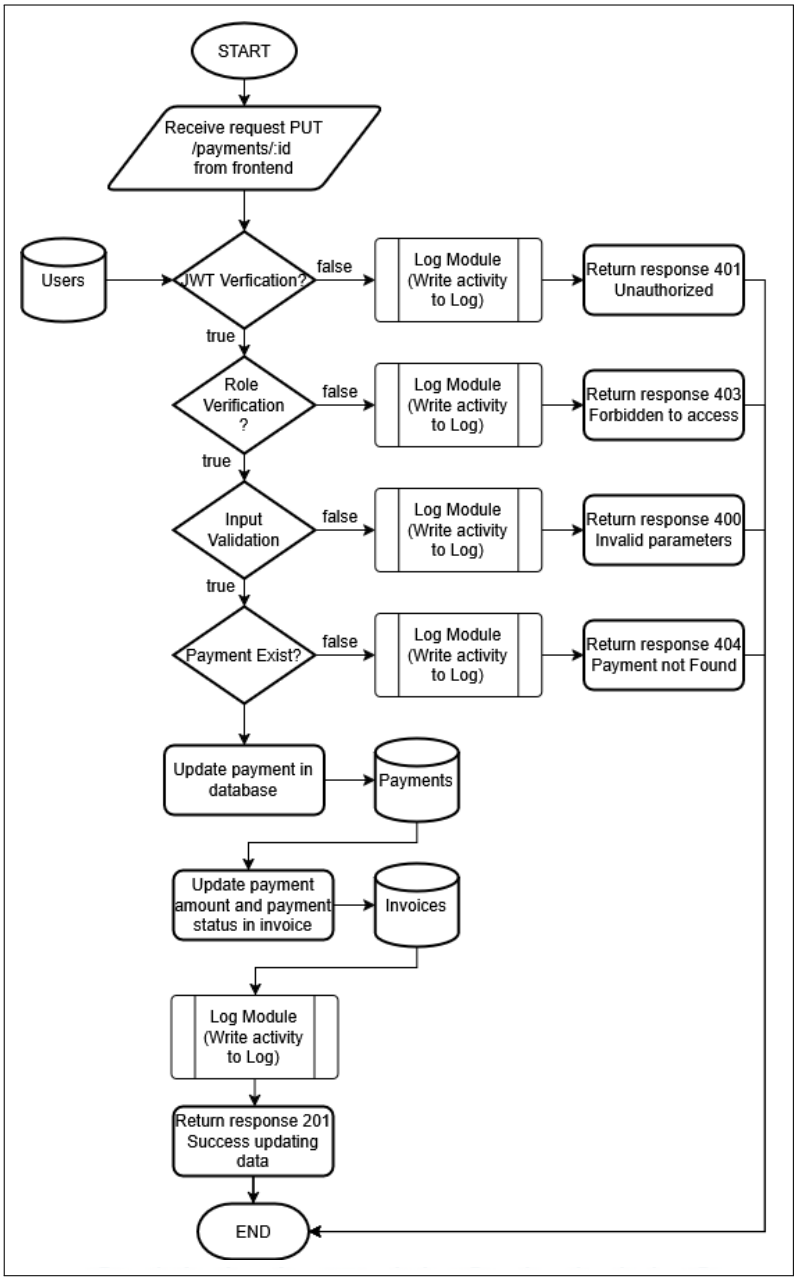
Invoice sudah lunas. Jika belum, program akan memasukkan data *payment* yang baru di *database* lalu melakukan *update* informasi di *Invoice* dengan jumlah yang sesuai dengan di *database payment.* Terakhir, aktivitas dicatat dalam *log* dan *server* mengembalikan respons bahwa *Create Payment* sukses.



Gambar 3.7. *Flowchart - Create Payment*

B *Update Payment*

Flowchart ini menggambarkan alur untuk melakukan modifikasi data pembayaran yang ada pada *database*. Proses dimulai ketika *server* mendapatkan *request* dengan *method PUT* dan *endpoint* berupa */payments/:id* dari *front-end*. Kemudian, *back-end* melakukan verifikasi akun, *role*, dan *input* yang diberikan oleh pengguna. Selanjutnya, sistem akan mengecek apakah *payment* yang ingin diubah ada di *database* atau tidak. Jika tidak ada, program akan memberikan respons bahwa *payment* tidak ditemukan. Jika ada, sistem akan melakukan *update* pada data *payment* sebelumnya dengan data yang baru. Lalu, informasi yang dipengaruhi oleh *payment* di *invoice* juga ikut diperbarui. Terakhir, aktivitas dicatat dalam *log* dan *server* mengembalikan respons bahwa *Update Payment* sukses.

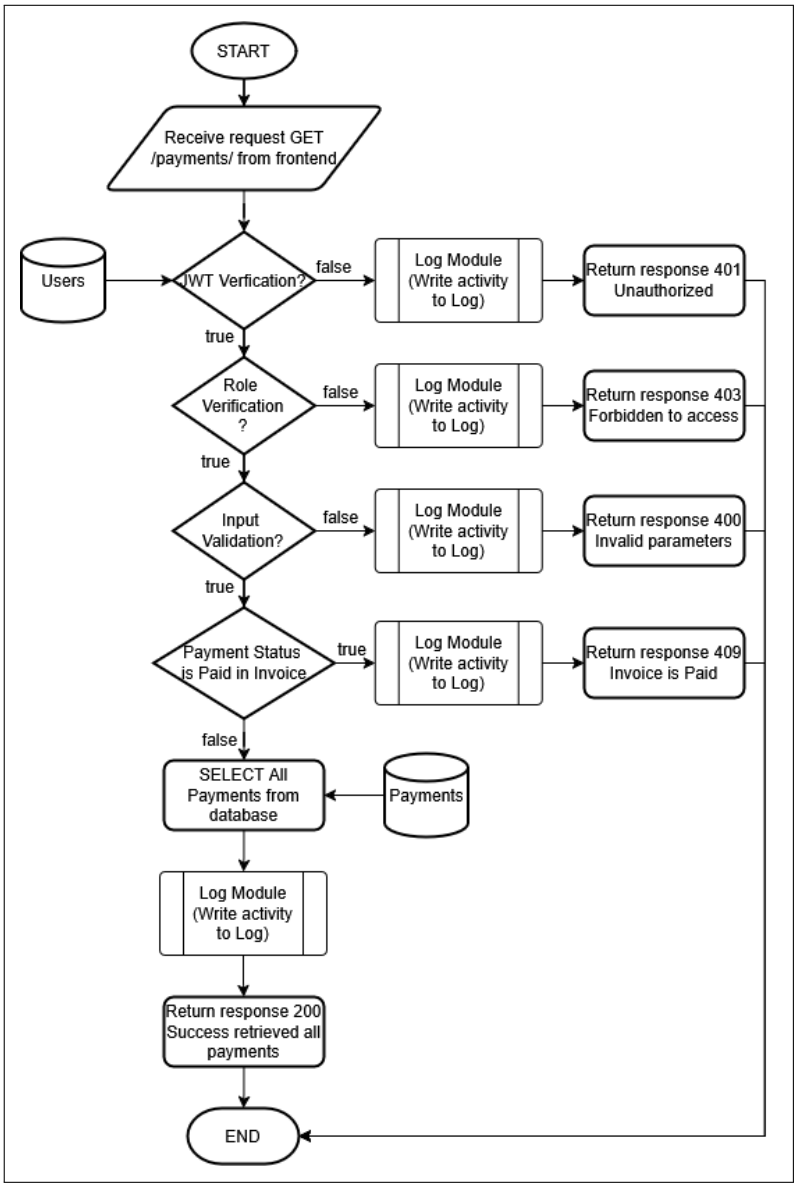


Gambar 3.8. *Flowchart - Update Payment*

C *Get All Payment*

Flowchart ini menggambarkan alur pemanggilan seluruh data *payment* dari *database.* Proses dimulai ketika *server* mendapatkan *request* dengan *method GET* dan *endpoint* berupa */payment* dari *front-end*. Kemudian, *back-end* melakukan verifikasi akun, *role*, dan *input* yang diberikan oleh pengguna. Selanjutnya, sistem

akan melakukan *query* pada *database* untuk mengambil seluruh data *payment*. Terakhir, aktivitas dicatat dalam *log* dan *server* mengembalikan respons bahwa *Update Payment* sukses.

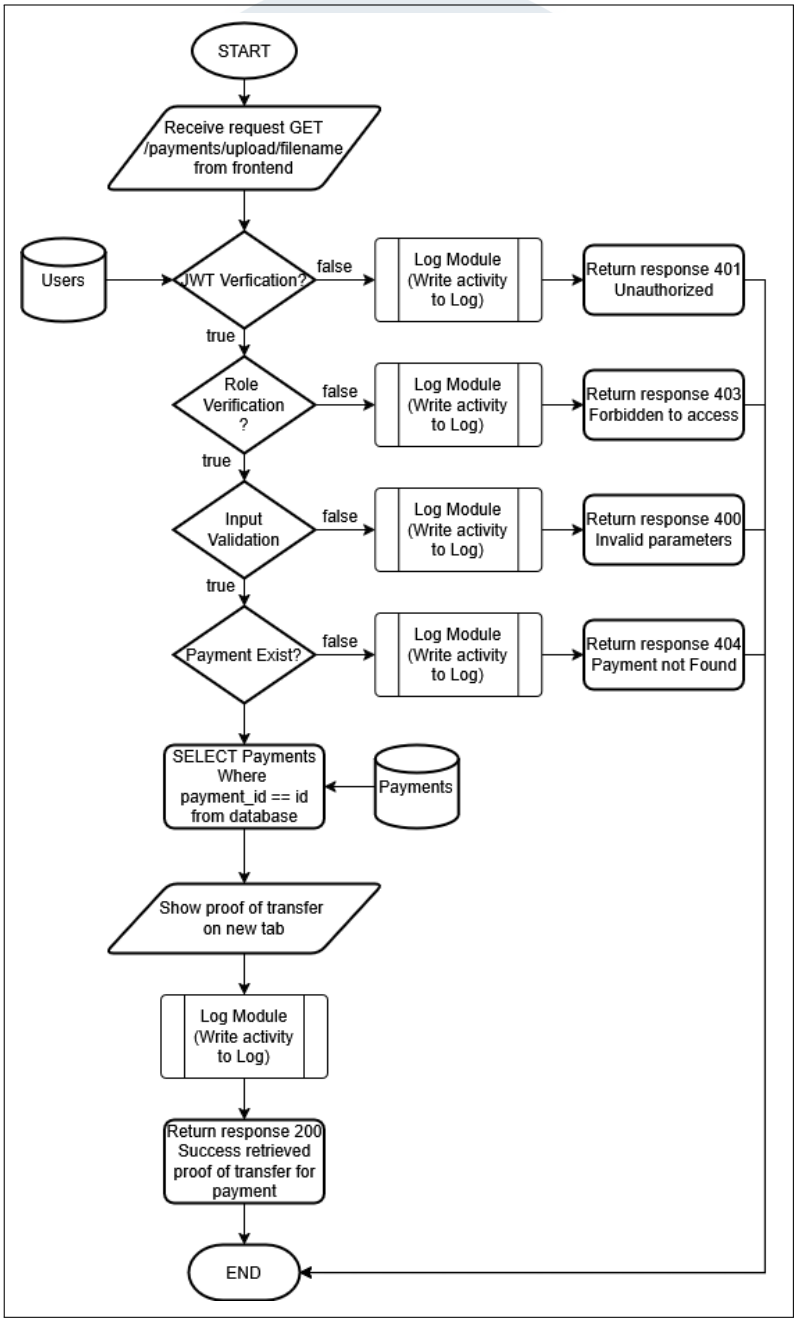


Gambar 3.9. *Flowchart - Get All Payment*

D *Get Payment by ID*

Flowchart ini menggambarkan alur pemanggilan data *payment* berdasarkan ID yang dicari di *database.* Proses dimulai ketika *server* mendapatkan *request* dengan *method GET* dan *endpoint* berupa */payment/:id* dari *front-end.* Kemudian,

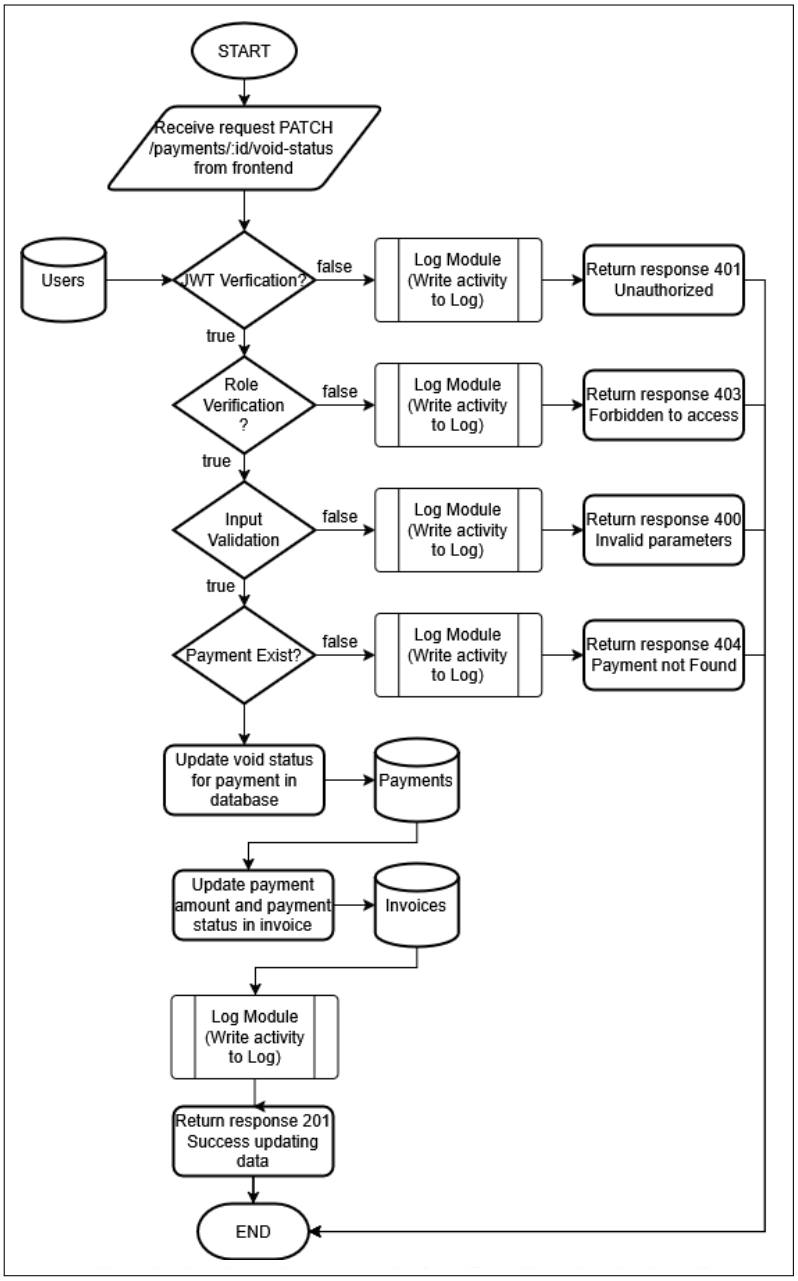
back-end melakukan verifikasi akun, *role*, dan *input* yang diberikan oleh pengguna. Selanjutnya, sistem akan mengecek apakah *payment* yang dicari ada atau tidak. Jika ada, program akan mengambil data *payment* yang memiliki ID yang sama dengan *payment* yang dicari. Terakhir, aktivitas dicatat dalam *log* dan *server* mengembalikan respons bahwa *Update Payment* sukses.



Gambar 3.10. *Flowchart - Get Payment by ID*

E *Toggle Void Status for Payment*

Flowchart ini menggambarkan alur untuk mengubah status pembayaran berlaku atau tidak. Proses dimulai ketika *server* mendapatkan *request* dengan *method PATCH* dan *endpoint* berupa */payments/:id/void-status* dari *front-end*. Kemudian, *back-end* melakukan verifikasi akun, *role*, dan *input* yang diberikan oleh pengguna. Selanjutnya, sistem akan mengecek apakah *payment* yang ingin diubah ada di *database* atau tidak. Jika tidak ada, program akan memberikan respons bahwa *payment* tidak ditemukan. Jika ada, sistem akan mengubah status *void* pada *payment*. Lalu, informasi yang dipengaruhi oleh *payment* di *invoice* juga ikut diperbarui. Terakhir, aktivitas dicatat dalam *log* dan *server* mengembalikan respons bahwa *Update Payment* sukses.

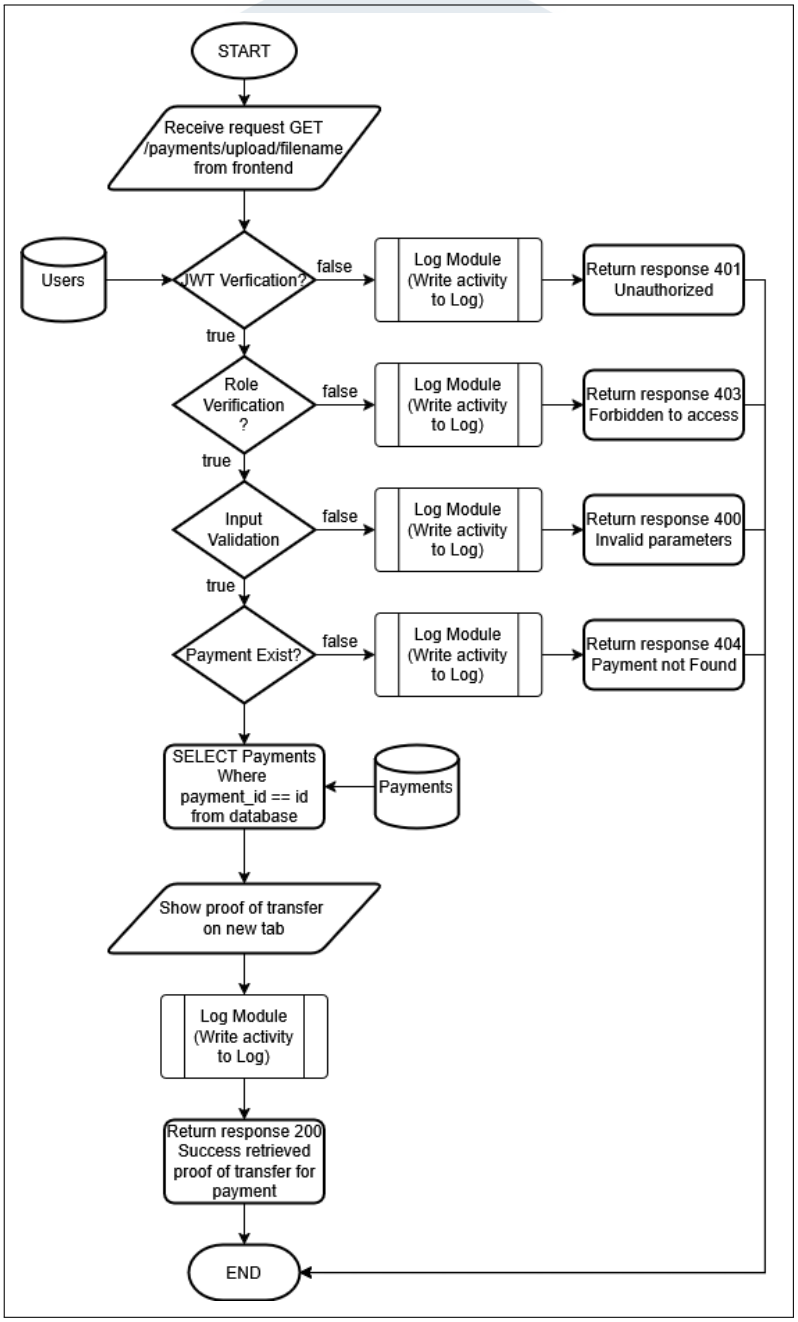


Gambar 3.11. *Flowchart - Toggle Void Status for Payment*

F *Get Proof of Transfer from Payment*

Flowchart ini menggambarkan alur untuk melihat *file* bukti pembayaran yang diunggah. Proses dimulai ketika server mendapatkan *request* dengan *method GET* dan *endpoint* berupa */payment/upload/filename* dari *front-end*. Kemudian, *back-end* melakukan verifikasi akun, *role*, dan *input* yang diberikan oleh pengguna.

Selanjutnya, sistem akan mengecek apakah *payment* yang dicari ada atau tidak. Jika ada, program akan mengambil data *payment* yang memiliki ID yang sama dengan *payment* yang dicari. Setelah *file path* didapat, *browser* akan menampilkan *file* bukti pembayaran di tab baru pada *browser*. Terakhir, aktivitas dicatat dalam *log* dan *server* mengembalikan respons bahwa *Update Payment* sukses.

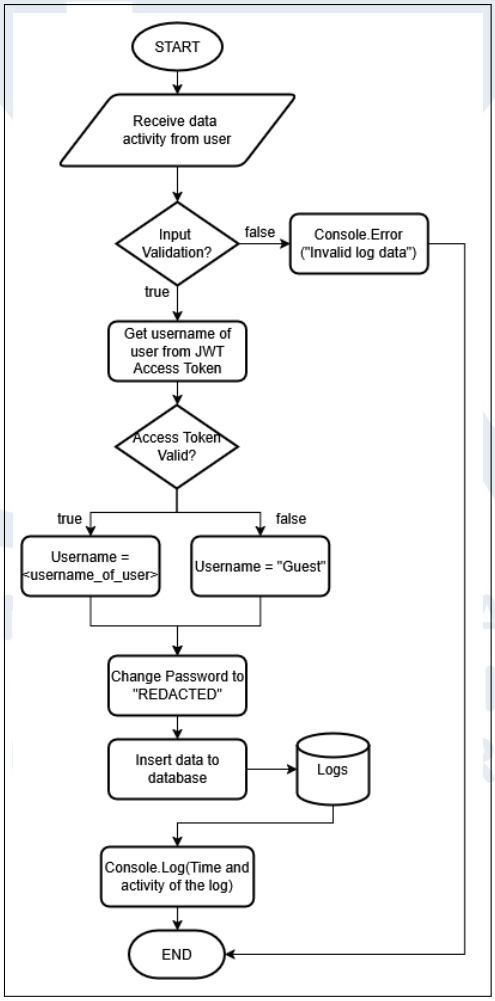


Gambar 3.12. *Flowchart - Get Proof of Transfer from Payment*

G *Log Module*

Flowchart ini menggambarkan alur pencatatan aktivitas yang terjadi pada *website*. Proses dimulai ketika *server* mendapatkan *request* dari *front-end*. Kemudian, *back-end* akan melakukan validasi *input* yang masuk. Jika *input* tidak valid, akan muncul `console.error()` pada terminal *server*.

Setelah itu jika valid, sistem akan mencari *username* yang melakukan *request* dari *access token* yang mereka punya. Jika punya akses *token*, *username* yang tersimpan adalah data yang berhasil dilakukan proses *decoding*. Sebaliknya, jika tidak valid, *username* yang disimpan dengan nama *"Guest."* Setelah itu data yang berisikan informasi *password* diubah, baru *log* bisa tersimpan dalam *database.* Setelah semua selesai, program akan menampilkan `console.log()` di terminal *server*.



Gambar 3.13. *Flowchart - Log Module*

3.4.5 *Database Schema*

Pada bagian ini, skema *database* dibuat sebagai rancangan dari *database* yang akan dibuat. Gambar 3.14 menunjukkan relasi dan data apa saja yang akan disimpan di tiap tabel. Tabel yang akan dibuat antara lain tabel *users*, *clients*, *invoices*, *invoice details*, *payments*, dan *logs*.

Gambar 3.14. Skema *database website invoicing*

Untuk lebih jelas lagi, selanjutnya akan dijelaskan mengenai struktur tabel dari sistem *invoicing* beserta keterangan pada tiap kolom.

A Tabel *Users*

Tabel 3.2 menjelaskan struktur dari tabel *users* yang digunakan untuk menyimpan data terkait *user* yang mengakses *website*. Data yang disimpan berupa `user_id`, `username`, `password`, `role`, dan keterangan kapan data *user* dibuat atau

dimodifikasi.

Tabel 3.2. Tabel *Users*

Kolom	Tipe Data	Keterangan
`user_id`	`UUID (PRIMARY KEY)`	ID unik pengguna. Digunakan sebagai *primary key* di *database.*
`username`	`String (UNIQUE)`	*Username* unik yang digunakan ketika *login*.
`password`	`String`	Kata sandi.
`role`	`String`	*Role* pengguna.
`deleted_at`	`DateTime (NULLABLE)`	Penanda jika data dihapus (*soft delete.*)
`created_at`	`DateTime`	Waktu pembuatan.
`updated_at`	`DateTime`	Waktu terakhir diubah.

3.4.6 Tabel *Clients*

Tabel 3.3 menjelaskan struktur dari tabel *clients* yang digunakan untuk menyimpan data terkait klien dari perusahaan *website*. Data yang disimpan berupa `client_id`, nama klien, mata uang yang digunakan oleh klien, negara dari klien, kode pos, nomor telepon dan keterangan kapan data klien dibuat atau dimodifikasi.

Tabel 3.3. Tabel *Clients*

Kolom	Tipe Data	Keterangan
client_id	UUID (PRIMARY KEY)	ID unik klien. Digunakan sebagai *primary key* di *database.*
client_name	String (UNIQUE)	Nama klien.
currency	String	Mata uang yang digunakan sebagai pembayaran.
country	String	Negara klien.
client_address	String	Alamat lengkap perusahaan klien.
postal_code	String	Kode pos perusahaan klien.
client_phone	String	Nomor telepon klien.
deleted_at	DateTime (NULLABLE)	Penanda jika data dihapus (*soft delete*)
created_at	DateTime	Waktu pembuatan
updated_at	DateTime	Waktu terakhir diubah

3.4.7 Tabel *Invoices*

Tabel 3.4 menjelaskan struktur dari tabel *invoices* yang digunakan untuk menyimpan data terkait *invoices* untuk klien. Data yang disimpan berupa invoice_id, nomor *invoice*, tanggal *invoice* berlaku, tanggal jatuh tempo, jumlah tagihan yang harus dibayar, status pembayaran dan keterangan kapan data *invoice* dibuat atau dimodifikasi.

Tabel 3.4. Tabel *Invoices*

Kolom	Tipe Data	Keterangan
invoice_id	UUID (PRIMARY KEY)	ID unik *invoice*. Digunakan sebagai *primary key* di *database.*
invoice_number	String (UNIQUE)	Nomor *invoice*.
issue_date	DateTime	Tanggal *invoice* mulai berlaku.
due_date	DateTime	Tanggal *invoice* jatuh tempo.
sub_total	Float	Total yang harus dibayar sebelum ditambah dengan pajak.
tax_rate	Float	Persentase pajak.
tax_amount	Float	Jumlah pajak yang didapat setelah dikalkulasi.
total	Float	Total tagihan.
tax_invoice_number	String	Nomor faktur pajak.
amount_paid	Float	Total yang sudah dibayarkan.
payment_status	String	Status pembayaran.
voided_at	DateTime (NULLABLE)	Status *invoice* berlaku atau tidak.
created_at	DateTime	Waktu pembuatan.
updated_at	DateTime	Waktu terakhir diubah.
client_id	UUID (FOREIGN KEY)	Kolom yang digunakan untuk membuat relasi ke *clients*. Relasi yang dibuat adalah *1-to-many* (*client* $\rightarrow$ *invoice*)

3.4.8 Tabel *Invoice Details*

Invoice detail merupakan tabel yang menyimpan informasi detail tentang tagihan yang akan diberikan pada klien. Satu *invoice* bisa memiliki lebih dari satu *invoice detail*. Data yang disimpan berupa detail penggunaan oleh klien. Di antaranya adalah invoice_detail_id, catatan transaksi, jumlah pengiriman, total harga, dan keterangan kapan data *invoice* dibuat atau dimodifikasi.

Tabel 3.5. Tabel *Invoice Details*

Kolom	Tipe Data	Keterangan
invoice_detail_id	UUID (PRIMARY KEY)	ID unik *invoice detail*. Digunakan sebagai *primary key* di *database.*
transaction_note	String (NULLABLE)	Catatan transaksi jika ada.
delivery_count	Int	Jumlah pengiriman yang digunakan oleh klien.
price_per_delivery	Float	Harga per pengiriman.
amount	Float	Total nilai baris
deleted_at	DateTime (NULLABLE)	Penanda jika data dihapus (*soft delete*)
created_at	DateTime	Waktu pembuatan
updated_at	DateTime	Waktu terakhir diubah
invoice_id	UUID (FOREIGN KEY)	Kolom yang digunakan untuk membuat relasi ke *invoices*. Relasi yang dibuat adalah *1-to-many* (*invoices* → *invoice_details*)

3.4.9 Tabel *Payments*

Tabel 3.6 menjelaskan struktur dari tabel *payments* yang digunakan untuk menyimpan data terkait pembayaran yang dilakukan oleh klien. Data yang disimpan berupa payment_id, tanggal pembayaran, jumlah yang dibayar, bukti transfer, status pembayaran berlaku atau tidak dan keterangan kapan data *payment* dibuat atau dimodifikasi.

Tabel 3.6. Tabel *Payments*

Kolom	Tipe Data	Keterangan
payment_id	UUID (PRIMARY KEY)	ID unik pembayaran. Digunakan sebagai *primary key* di *database.*
payment_date	DateTime	Tanggal pembayaran masuk.
amount_paid	Float	Nominal pembayaran yang masuk.
proof_of_transfer	String	Bukti transfer pembayaran.
voided_at	DateTime (NULLABLE)	Status *payment* berlaku atau tidak.
created_at	DateTime	Waktu pembuatan.
updated_at	DateTime	Waktu terakhir diubah.
invoice_id	UUID (FOREIGN KEY)	Kolom yang digunakan untuk membuat relasi ke tabel *invoices*. Relasi yang dibuat adalah *1-to-many* (*invoices* $\rightarrow$ *payments*)
invoice_number	String	Informasi tambahan untuk mempermudah dalam mengetahui pembayaran ditujukan ke nomor *invoice* berapa.

3.4.10 Tabel *Logs*

Tabel 3.7 menjelaskan struktur dari tabel *logs* yang digunakan untuk menyimpan data aktivitas yang pernah terjadi pada *website*. Data yang disimpan berupa log_id, *ip address* pengguna, akses *token user*, *method* yang digunakan saat *request* ke *server*, *endpoint* yang coba diakses, *payload* yang dikirim ke *server*, status dan respons dari *server*, serta keterangan kapan data *log* dibuat.

Tabel 3.7. Tabel *Logs*

Kolom	Tipe Data	Keterangan
log_id	Integer (PRIMARY KEY, AUTO INCREMENT)	ID *log*.
ip	String	Alamat IP pengguna.
acces_token	String (NULLABLE)	Akses *token* yang dimiliki *pengguna*
username	String	*Username* pengguna
method	String	Jenis HTTP *method* (GET, POST, PUT, dll.)
endpoint	String	*Endpoint* yang diakses.
payload	JSON (NULLABLE)	Parameter *request* yang dikirim oleh *pengguna* ke *server*.
status	String	Status eksekusi *(success/error)*
status_message	String	Respons pesan yang dikirim oleh *server* ke pengguna.
created_at	DateTime	Waktu *log* dicatat.

3.5 Implementasi Sistem

Bagian ini menjelaskan proses implementasi sistem *back-end* dan integrasi dengan *front-end* dalam pengembangan sistem *invoicing* berbasis *web*. Implementasi dilakukan menggunakan *framework* Express.js untuk sisi *back-end*, dan integrasi dilakukan dengan *front-end* pada *framework* React.js menggunakan RESTful API berbasis protokol HTTP.

3.5.1 Pembuatan Fitur *Back-end*

Pada tahap ini, pengembangan dilakukan untuk fitur-fitur *back-end* modul pembayaran. Fitur ini diantaranya ada pembuatan, *update*, pengambilan, dan pengelolaan status dari data pembayaran. Implementasi *back-end* dilakukan menggunakan pendekatan RESTful API yang memudahkan integrasi dengan *front-end*. Selain itu, fitur *log* juga dibuat untuk mencatat segala aktivitas yang terjadi.

Seperti yang bisa dilihat pada gambar 3.15, gambar tersebut menjelaskan rute *endpoint* mana saja yang tersedia bagi modul *payment.*

```
router.get( path: '/', authGuard, roleGuard(['finance', 'management']), getAllPaymentController);
router.get( path: '/:id', authGuard, roleGuard(['finance', 'management']), getPaymentByIdController);

router.get(
  path: '/upload/:filename',
  authGuard,
  roleGuard(['finance', 'management']),
  getProofPaymentController,
);

router.post(
  path: '/',
  authGuard,
  roleGuard(['finance']),
  upload_payment.single( fieldName: 'proof_of_transfer'),
  createPaymentController,
);
router.put(
  path: '/:id',
  authGuard,
  roleGuard(['finance']),
  upload_payment.single( fieldName: 'proof_of_transfer'),
  editPaymentController,
);
router.patch(
  path: '/:id/void-status',
  authGuard,
  roleGuard(['finance', 'management']),
  togglePaymentVoidStatusController,
);
```

Gambar 3.15. Implementasi - Seluruh *Endpoint* dari *Payment*

A *Controller for Create Payment*

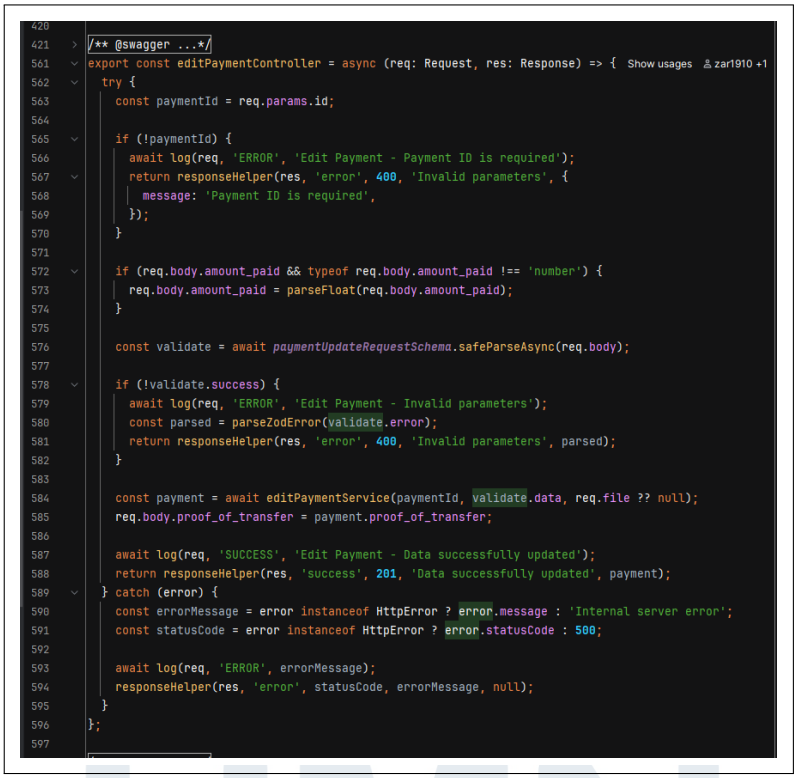
Fungsi ini bertanggung jawab dalam menerima permintaan dari *front-end* dan menyimpan data pembayaran ke *database*.

```
/** @swagger ...*/
export const createPaymentController = async (req: Request, res: Response): Promise<void> => {  Show usages   zar1910 +1
  try {
    if (req.body.amount_paid && typeof req.body.amount_paid !== 'number') {
      req.body.amount_paid = parseFloat(req.body.amount_paid);
    }

    if (!req.body.proof_of_transfer && req.file) {
      req.body.proof_of_transfer = `payments/upload/${req.file.filename}`;
    }

    const validate = await paymentRequestSchema.safeParseAsync(req.body);

    if (!validate.success) {
      await log(req, 'ERROR', 'Create Payment - Invalid parameters');
      const parsedError = parseZodError(validate.error);
      return responseHelper(res, 'error', 400, 'Invalid parameters', parsedError);
    }

    const payment = await createPaymentService(validate.data, req.file!);
    req.body.proof_of_transfer = payment.proof_of_transfer;

    await log(req, 'SUCCESS', 'Create Payment - Data successfully created');
    return responseHelper(res, 'success', 201, 'Data successfully created', payment);
  } catch (error) {
    const errorMessage = error instanceof HttpError ? error.message : 'Internal server error';
    const statusCode = error instanceof HttpError ? error.statusCode : 500;

    await log(req, 'ERROR', errorMessage);
    responseHelper(res, 'error', statusCode, errorMessage, null);
  }
};
```

Gambar 3.16. Implementasi - *Create Payment Controller*

B *Controller for Update Payment*

Fitur ini memungkinkan untuk melakukan *update* informasi pembayaran yang sudah ada di *database*.

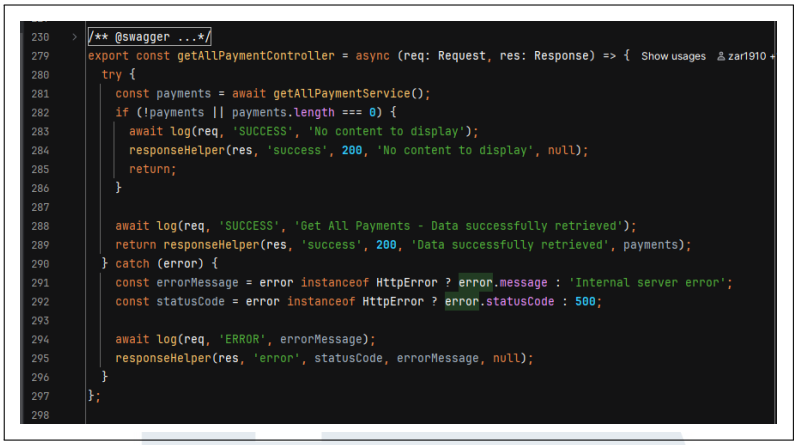


```
/** @swagger ...*/
export const editPaymentController = async (req: Request, res: Response) => {  Show usages  zar1910 +1
  try {
    const paymentId = req.params.id;

    if (!paymentId) {
      await log(req, 'ERROR', 'Edit Payment - Payment ID is required');
      return responseHelper(res, 'error', 400, 'Invalid parameters', {
        message: 'Payment ID is required',
      });
    }

    if (req.body.amount_paid && typeof req.body.amount_paid !== 'number') {
      req.body.amount_paid = parseFloat(req.body.amount_paid);
    }

    const validate = await paymentUpdateRequestSchema.safeParseAsync(req.body);

    if (!validate.success) {
      await log(req, 'ERROR', 'Edit Payment - Invalid parameters');
      const parsed = parseZodError(validate.error);
      return responseHelper(res, 'error', 400, 'Invalid parameters', parsed);
    }

    const payment = await editPaymentService(paymentId, validate.data, req.file ?? null);
    req.body.proof_of_transfer = payment.proof_of_transfer;

    await log(req, 'SUCCESS', 'Edit Payment - Data successfully updated');
    return responseHelper(res, 'success', 201, 'Data successfully updated', payment);
  } catch (error) {
    const errorMessage = error instanceof HttpError ? error.message : 'Internal server error';
    const statusCode = error instanceof HttpError ? error.statusCode : 500;

    await log(req, 'ERROR', errorMessage);
    responseHelper(res, 'error', statusCode, errorMessage, null);
  }
};
```

Gambar 3.17. Implementasi - *Update Payment Controller*

C *Controller for Get All Payment*

Controller ini berfungsi untuk mengambil seluruh data pembayaran dari *database* dan menampilkannya ke pengguna.

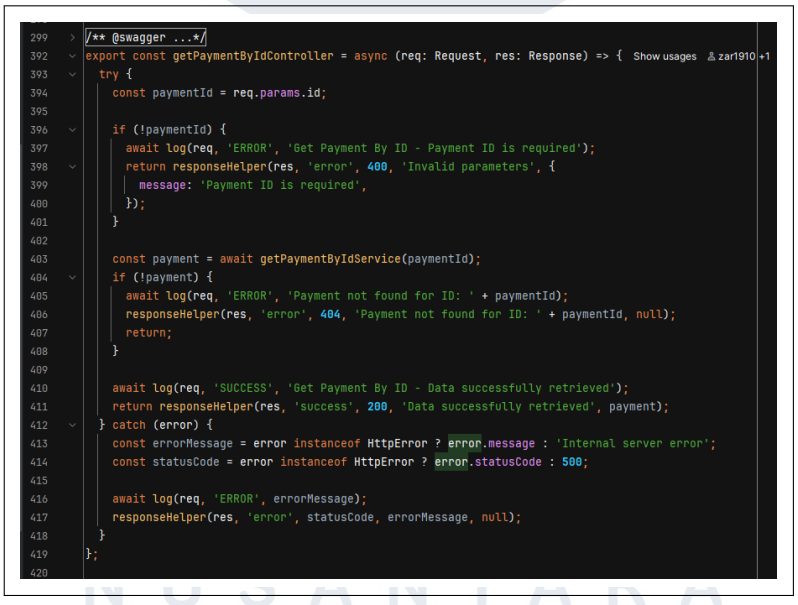


```
/** @swagger ...*/
export const getAllPaymentController = async (req: Request, res: Response) => {  Show usages  zar1910 +
  try {
    const payments = await getAllPaymentService();
    if (!payments || payments.length === 0) {
      await log(req, 'SUCCESS', 'No content to display');
      responseHelper(res, 'success', 200, 'No content to display', null);
      return;
    }

    await log(req, 'SUCCESS', 'Get All Payments - Data successfully retrieved');
    return responseHelper(res, 'success', 200, 'Data successfully retrieved', payments);
  } catch (error) {
    const errorMessage = error instanceof HttpError ? error.message : 'Internal server error';
    const statusCode = error instanceof HttpError ? error.statusCode : 500;

    await log(req, 'ERROR', errorMessage);
    responseHelper(res, 'error', statusCode, errorMessage, null);
  }
};
```

Gambar 3.18. Implementasi - *Get All Payment Controller*

D *Controller for Get Payment by ID*

Endpoint ini digunakan untuk mengambil satu data pembayaran yang dicari berdasarkan ID yang ingin dilihat.

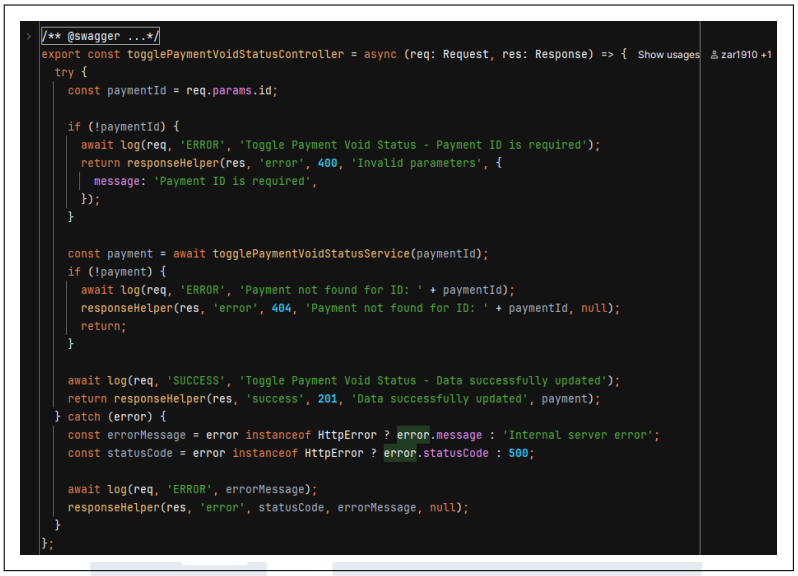


```
/** @swagger ...*/
export const getPaymentByIdController = async (req: Request, res: Response) => {  Show usages  zar1910 +1
  try {
    const paymentId = req.params.id;

    if (!paymentId) {
      await log(req, 'ERROR', 'Get Payment By ID - Payment ID is required');
      return responseHelper(res, 'error', 400, 'Invalid parameters', {
        message: 'Payment ID is required',
      });
    }

    const payment = await getPaymentByIdService(paymentId);
    if (!payment) {
      await log(req, 'ERROR', 'Payment not found for ID: ' + paymentId);
      responseHelper(res, 'error', 404, 'Payment not found for ID: ' + paymentId, null);
      return;
    }

    await log(req, 'SUCCESS', 'Get Payment By ID - Data successfully retrieved');
    return responseHelper(res, 'success', 200, 'Data successfully retrieved', payment);
  } catch (error) {
    const errorMessage = error instanceof HttpError ? error.message : 'Internal server error';
    const statusCode = error instanceof HttpError ? error.statusCode : 500;

    await log(req, 'ERROR', errorMessage);
    responseHelper(res, 'error', statusCode, errorMessage, null);
  }
};
```

Gambar 3.19. Implementasi - *Get Payment by ID Controller*

E *Controller for Toggle Void Status in Payment*

Fitur ini bertugas untuk mengubah status pembayaran menjadi *"void"* atau valid atau tidaknya pembayaran tersebut.



```
/** @swagger ...*/
export const togglePaymentVoidStatusController = async (req: Request, res: Response) => {  Show usages  zar1910 +1
  try {
    const paymentId = req.params.id;

    if (!paymentId) {
      await log(req, 'ERROR', 'Toggle Payment Void Status - Payment ID is required');
      return responseHelper(res, 'error', 400, 'Invalid parameters', {
        message: 'Payment ID is required',
      });
    }

    const payment = await togglePaymentVoidStatusService(paymentId);
    if (!payment) {
      await log(req, 'ERROR', 'Payment not found for ID: ' + paymentId);
      responseHelper(res, 'error', 404, 'Payment not found for ID: ' + paymentId, null);
      return;
    }

    await log(req, 'SUCCESS', 'Toggle Payment Void Status - Data successfully updated');
    return responseHelper(res, 'success', 201, 'Data successfully updated', payment);
  } catch (error) {
    const errorMessage = error instanceof HttpError ? error.message : 'Internal server error';
    const statusCode = error instanceof HttpError ? error.statusCode : 500;

    await log(req, 'ERROR', errorMessage);
    responseHelper(res, 'error', statusCode, errorMessage, null);
  }
};
```

Gambar 3.20. Implementasi - *Toggle Void Status Controller*

F *Controller for Get Proof of Transfer from Payment*

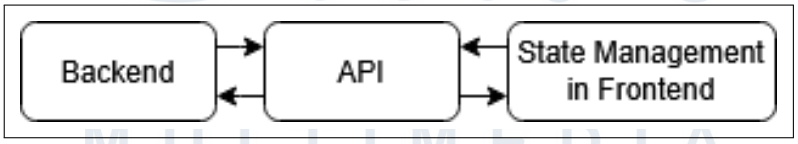
Fitur ini digunakan untuk melihat *file* pembayaran yang disimpan di *database*.

```
/** @swagger ...*/
export const getProofPaymentController = async (req: Request, res: Response) => {
  try {
    const filename = req.params.filename;

    if (!filename) {
      await log(req, 'ERROR', 'Get Proof of Payment - Filename is required');
      return responseHelper(res, 'error', 400, 'Invalid parameters', {
        message: 'Payment Filename is required',
      });
    }

    const filePath = await getProofPaymentService(filename);
    const safePath = path.resolve(filePath);
    const baseDir = path.resolve('uploads', 'payments');

    if (!safePath.startsWith(baseDir)) {...}

    if (!fs.existsSync(safePath)) {
      await log(req, 'ERROR', 'Get Proof of Payment - File not found');
      return responseHelper(res, 'error', 404, 'Data not found', { message: 'File not found' });
    }

    res.sendFile(safePath, async err => {
      if (err) {
        await log(req, 'ERROR', 'Get Proof of Payment - Error sending file');
        return responseHelper(res, 'error', 500, 'Internal server error', {
          error: 'Error sending file',
        });
      }
    });

    await log(req, 'SUCCESS', 'Get Proof of Payment - File successfully sent');
    return responseHelper(res, 'success', 200, 'Data successfully retrieved', { filePath });
  } catch (error) {
    const errorMessage = error instanceof HttpError ? error.message : 'Internal server error';
    const statusCode = error instanceof HttpError ? error.statusCode : 500;

    await log(req, 'ERROR', errorMessage);
    responseHelper(res, 'error', statusCode, errorMessage, null);
  }
};
```

Gambar 3.21. Implementasi - *Get Proof of Transfer Controller*

3.5.2 Integrasi *Front-end* dan *Back-end*

Proses integrasi dilakukan dengan memanfaatkan *endpoint* yang telah dibuat di *back-end*. Kemudian, *endpoint* tersebut dihubungkan ke *front-end* menggunakan HTTP *request*. Seluruh alur komunikasi antara *front-end* dan *back-end* divisualisasikan dalam gambar 3.22:

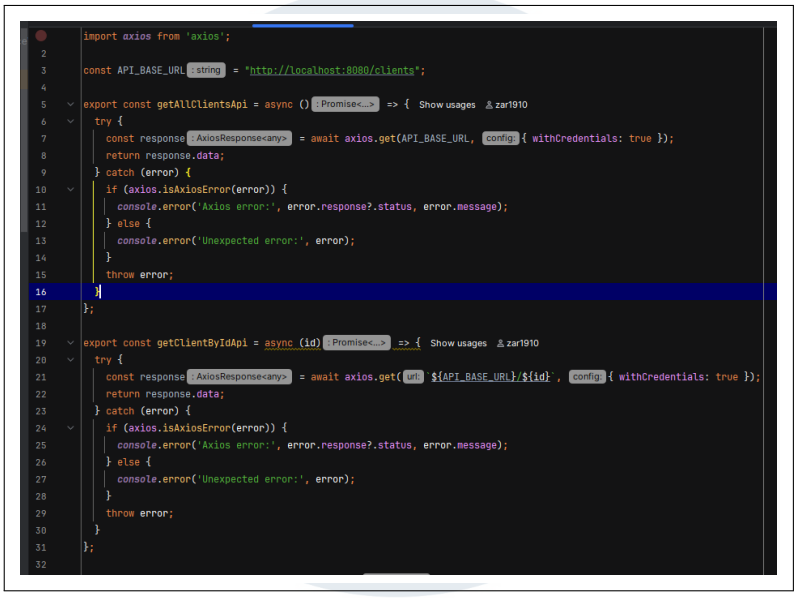


Gambar 3.22. Integrasi - Alur Komunikasi *Front-end* dan *Back-end*

Selanjutnya, proses integrasi dibagi menjadi empat tahap yaitu integrasi di *Client Page*, *Invoice Page*, *Payment Page*, dan *Report Page*

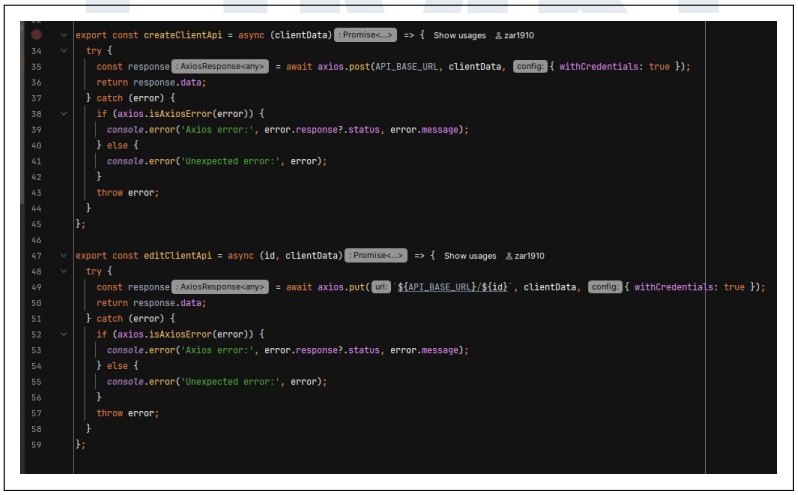
A *Client Page*

Pada halaman ini, *front-end* terhubung ke API untuk mengambil data *client* dan menyimpannya ke dalam *state management*. Fitur yang bisa dilakukan di *Client Page* antara lain *Create*, *Update*, *Get All Client*, dan *Get Client by ID*.



```
import axios from 'axios';

const API_BASE_URL: string = "http://localhost:8080/clients";

export const getAllClientsApi = async (): Promise<...> => {  Show usages  zar1910
  try {
    const response: AxiosResponse<any> = await axios.get(API_BASE_URL, config: { withCredentials: true });
    return response.data;
  } catch (error) {
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};

export const getClientByIdApi = async (id): Promise<...> => {  Show usages  zar1910
  try {
    const response: AxiosResponse<any> = await axios.get(url: `${API_BASE_URL}/${id}`, config: { withCredentials: true });
    return response.data;
  } catch (error) {
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};
```

Gambar 3.23. Integrasi - Implementasi API untuk *Client Page*



```
export const createClientApi = async (clientData): Promise<...> => {  Show usages  zar1910
  try {
    const response: AxiosResponse<any> = await axios.post(API_BASE_URL, clientData, config: { withCredentials: true });
    return response.data;
  } catch (error) {
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};

export const editClientApi = async (id, clientData): Promise<...> => {  Show usages  zar1910
  try {
    const response: AxiosResponse<any> = await axios.put(url: `${API_BASE_URL}/${id}`, clientData, config: { withCredentials: true });
    return response.data;
  } catch (error) {
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};
```

Gambar 3.24. Integrasi - Lanjutan Implementasi API untuk *Client Page*

```
export const createNewClient = createAsyncThunk('clients/createClient',
  async (clientData, { rejectWithValue }) => {
    try {
      const response = await createClientApi(clientData);
      console.log("Created new client:", response);
      return response;
    } catch (error) {
      console.error("Error creating client:", error.response?.data || error.message);
      return rejectWithValue(error.response.data);
    }
});
```

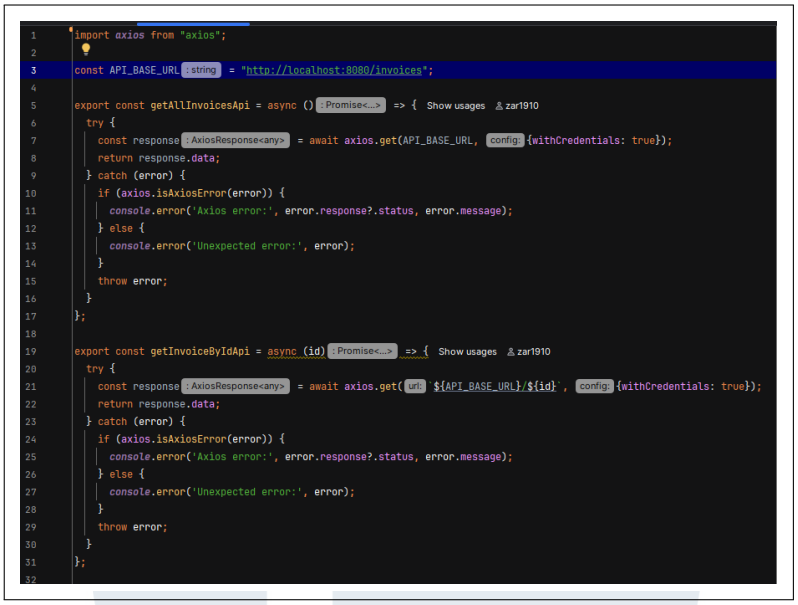
Gambar 3.25. Integrasi - *Passing* data dari API ke *state management* untuk *Client Page*

```
// Case for creating a new client
.addCase(createNewClient.pending, (state) => {
  state.loading = true;
  state.error = null;
  console.log("Creating new client pending...");
})
.addCase(createNewClient.fulfilled, (state, action) => {
  state.loading = false;
  state.allClients.push(action.payload.data);
  console.log("Created new client:", action.payload.data);
})
// Case for editing a client
.addCase(createNewClient.rejected, (state, action) => {
  state.loading = false;
  state.error = action.payload;
  console.log("Error creating client:", action.payload.data);
})
```

Gambar 3.26. Integrasi - Lanjutan *Passing* data dari API ke *state management* untuk *Client Page*

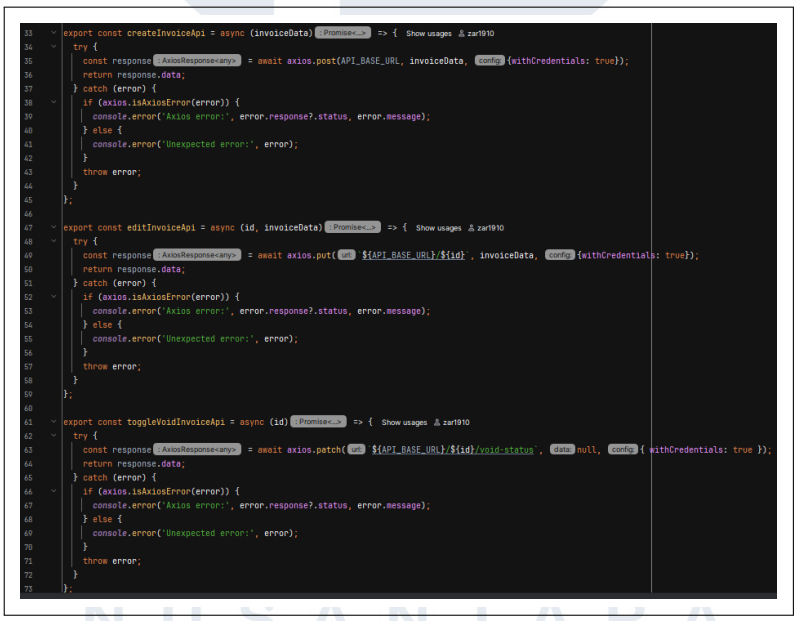
B *Invoice Page*

Halaman *invoice* terintegrasi dengan API yang mengambil data *invoice* dari *database*. Data kemudian diatur ke dalam *state management* untuk ditampilkan dalam *User Interface*.



```
import axios from "axios";

const API_BASE_URL : string = "http://localhost:8080/invoices";

export const getAllInvoicesApi = async () : Promise<...> => {  Show usages  zar1910
  try {
    const response : AxiosResponse<any> = await axios.get(API_BASE_URL, config: {withCredentials: true});
    return response.data;
  } catch (error) {
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};

export const getInvoiceByIdApi = async (id) : Promise<...> => {  Show usages  zar1910
  try {
    const response : AxiosResponse<any> = await axios.get( url: `${API_BASE_URL}/${id}`, config: {withCredentials: true});
    return response.data;
  } catch (error) {
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};
```

Gambar 3.27. Integrasi - Implementasi API untuk *Invoice Page*

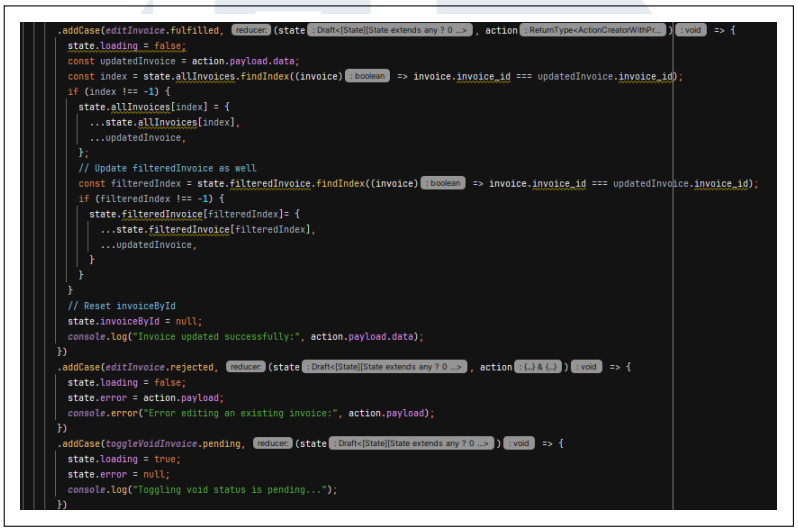


```
export const createInvoiceApi = async (invoiceData) : Promise<...> => {  Show usages  zar1910
  try {
    const response : AxiosResponse<any> = await axios.post(API_BASE_URL, invoiceData, config: {withCredentials: true});
    return response.data;
  } catch (error) {
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};

export const editInvoiceApi = async (id, invoiceData) : Promise<...> => {  Show usages  zar1910
  try {
    const response : AxiosResponse<any> = await axios.put( url: `${API_BASE_URL}/${id}`, invoiceData, config: {withCredentials: true});
    return response.data;
  } catch (error) {
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};

export const toggleVoidInvoiceApi = async (id) : Promise<...> => {  Show usages  zar1910
  try {
    const response : AxiosResponse<any> = await axios.patch( url: `${API_BASE_URL}/${id}/void-status`, data: null, config: { withCredentials: true });
    return response.data;
  } catch (error) {
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};
```

Gambar 3.28. Integrasi - Lanjutan Implementasi API untuk *Invoice Page*

```
export const editInvoice = createAsyncThunk("invoice/updateInvoice",
  async ({ id, invoiceData }, { rejectWithValue }) => {
    try {
      const response = await editInvoiceApi(id, invoiceData);
      console.log("Invoice edited successfully:", response);
      return response;
    } catch (error) {
      console.error("Error editing invoice:", error.response?.data || error.message);
      return rejectWithValue(error.response.data);
    }
  });
```

Gambar 3.29. Integrasi - *Passing* data dari API ke *state management* untuk *Invoice Page*

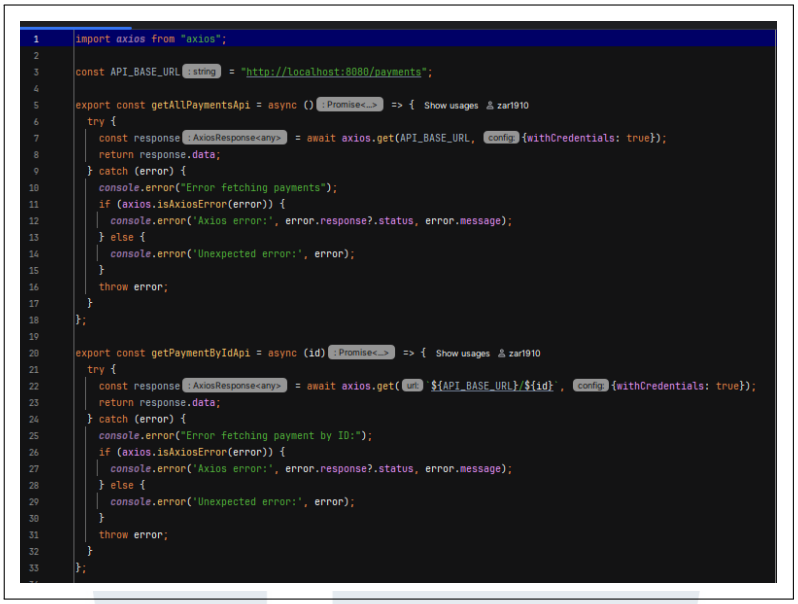


```
.addCase(editInvoice.fulfilled, (state, action) => {
  state.loading = false;
  const updatedInvoice = action.payload.data;
  const index = state.allInvoices.findIndex((invoice) => invoice.invoice_id === updatedInvoice.invoice_id);
  if (index !== -1) {
    state.allInvoices[index] = {
      ...state.allInvoices[index],
      ...updatedInvoice,
    };
    // Update filteredInvoice as well
    const filteredIndex = state.filteredInvoice.findIndex((invoice) => invoice.invoice_id === updatedInvoice.invoice_id);
    if (filteredIndex !== -1) {
      state.filteredInvoice[filteredIndex]= {
        ...state.filteredInvoice[filteredIndex],
        ...updatedInvoice,
      }
    }
  }
  // Reset invoiceById
  state.invoiceById = null;
  console.log("Invoice updated successfully:", action.payload.data);
})
.addCase(editInvoice.rejected, (state, action) => {
  state.loading = false;
  state.error = action.payload;
  console.error("Error editing an existing invoice:", action.payload);
})
.addCase(toggleVoidInvoice.pending, (state) => {
  state.loading = true;
  state.error = null;
  console.log("Toggling void status is pending...");
})
```

Gambar 3.30. Integrasi - Lanjutan *Passing* data dari API ke *state management* untuk *Invoice Page*

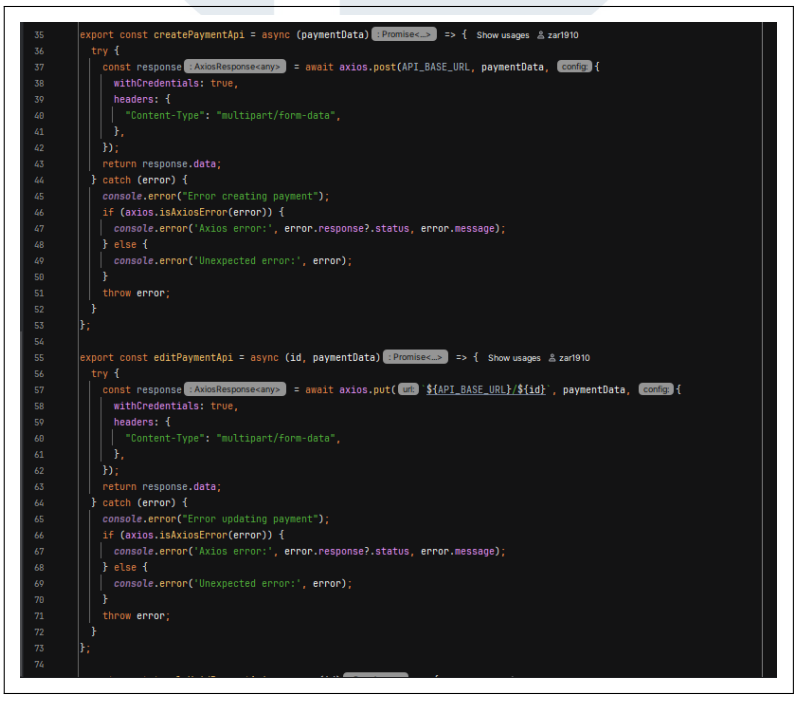
C *Payment Page*

Halaman *payment* bertugas untuk menangani pengambilan dan pengiriman data pembayaran melalui API. Seluruh proses seperti pembuatan, pembaruan, dan perubahan status pembayaran diatur melalui API dan disimpan ke dalam *state*.



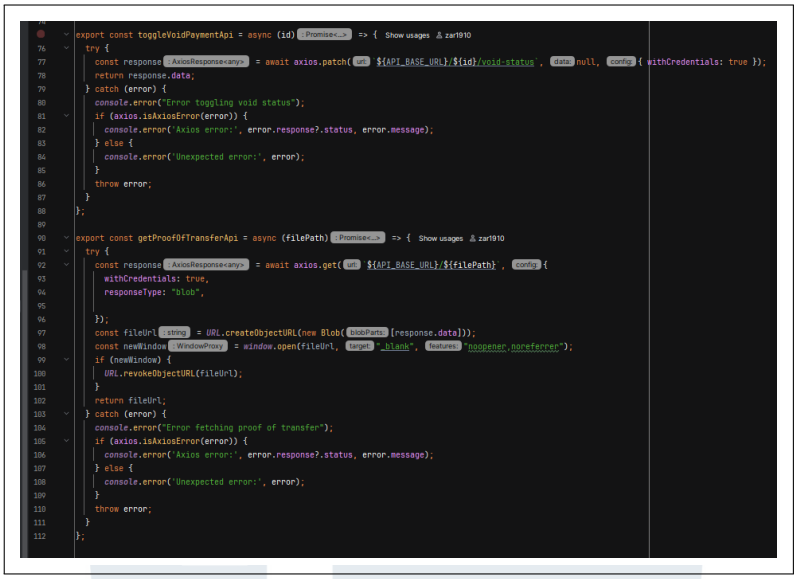
```
import axios from "axios";

const API_BASE_URL : string  = "http://localhost:8080/payments";

export const getAllPaymentsApi = async () : Promise<...>  => {  Show usages   zar1910
  try {
    const response : AxiosResponse<any>  = await axios.get(API_BASE_URL,  config: {withCredentials: true});
    return response.data;
  } catch (error) {
    console.error("Error fetching payments");
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};

export const getPaymentByIdApi = async (id) : Promise<...>  => {  Show usages   zar1910
  try {
    const response : AxiosResponse<any>  = await axios.get( url: `${API_BASE_URL}/${id}` ,  config: {withCredentials: true});
    return response.data;
  } catch (error) {
    console.error("Error fetching payment by ID:");
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};
```

Gambar 3.31. Integrasi - Implementasi API untuk *Payment Page*



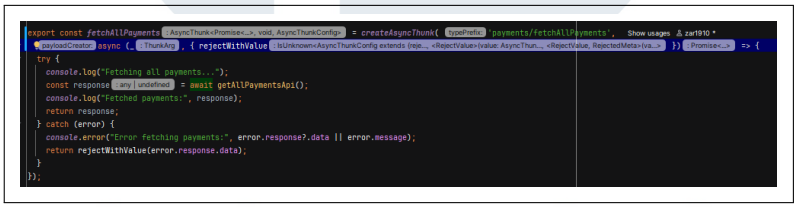
```
export const createPaymentApi = async (paymentData) : Promise<...>  => {  Show usages   zar1910
  try {
    const response : AxiosResponse<any>  = await axios.post(API_BASE_URL, paymentData,  config: {
      withCredentials: true,
      headers: {
        "Content-Type": "multipart/form-data",
      },
    });
    return response.data;
  } catch (error) {
    console.error("Error creating payment");
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};

export const editPaymentApi = async (id, paymentData) : Promise<...>  => {  Show usages   zar1910
  try {
    const response : AxiosResponse<any>  = await axios.put( url: `${API_BASE_URL}/${id}` , paymentData,  config: {
      withCredentials: true,
      headers: {
        "Content-Type": "multipart/form-data",
      },
    });
    return response.data;
  } catch (error) {
    console.error("Error updating payment");
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};
```

Gambar 3.32. Integrasi - Lanjutan Implementasi *API* untuk *Payment Page*



```
export const toggleVoidPaymentApi = async (id) => {
  try {
    const response = await axios.patch(`${API_BASE_URL}/${id}/void-status`, null, { withCredentials: true });
    return response.data;
  } catch (error) {
    console.error("Error toggling void status");
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};

export const getProofOfTransferApi = async (filePath) => {
  try {
    const response = await axios.get(`${API_BASE_URL}/${filePath}`, {
      withCredentials: true,
      responseType: "blob",

    });
    const fileUrl = URL.createObjectURL(new Blob([response.data]));
    const newWindow = window.open(fileUrl, "_blank", "noopener,noreferrer");
    if (newWindow) {
      URL.revokeObjectURL(fileUrl);
    }
    return fileUrl;
  } catch (error) {
    console.error("Error fetching proof of transfer");
    if (axios.isAxiosError(error)) {
      console.error('Axios error:', error.response?.status, error.message);
    } else {
      console.error('Unexpected error:', error);
    }
    throw error;
  }
};
```

Gambar 3.33. Integrasi - *Passing* data dari API ke *state management* untuk *Payment Page*



```
export const fetchAllPayments = createAsyncThunk("payments/fetchAllPayments", async (_, { rejectWithValue }) => {
  try {
    console.log("Fetching all payments...");
    const response = await getAllPaymentsApi();
    console.log("Fetched payments:", response);
    return response;
  } catch (error) {
    console.error("Error fetching payments:", error.response?.data || error.message);
    return rejectWithValue(error.response.data);
  }
});
```

Gambar 3.34. Integrasi - Lanjutan *Passing* data dari API ke *state management* untuk *Payment Page*

```
extraReducers: (builder) => {
  builder
    // Case for fetching all payments
    .addCase(fetchAllPayments.pending, (state) => {
      state.loading = true;
      state.error = null;
      console.log("Fetching pending...");
    })
    .addCase(fetchAllPayments.fulfilled, (state, action) => {
      state.loading = false;
      state.allPayments = action.payload?.data || [];
      state.filteredPayment = action.payload?.data || [];
      console.log("Fetched payments successfully:", action.payload);
    })
    .addCase(fetchAllPayments.rejected, (state, action) => {
      state.loading = false;
      state.error = action.payload;
      console.error("Error fetching payments:", action.payload);
    })
```

Gambar 3.35. Integrasi - Lanjutan ke-2 *Passing* data dari API ke *state management* untuk *Payment Page*

D *Report Page*

Halaman *report* digunakan untuk menampilkan data dari tabel *invoice* dan *payment* secara bersamaan. Karena data sudah tersedia dari kedua *endpoint* tersebut, tidak diperlukan pembuatan API baru. Data cukup diambil dari *state* dan ditampilkan dalam bentuk tabel atau grafik.

3.6 Hasil Pengetesan

Setelah tahap implementasi selesai, *website invoicing* coba dites kepada pengguna dari bagian manajemen dan keuangan. Pengetesan dilakukan dengan menggunakan skenario penggunaan yang sesuai dengan aktivitas operasional sehari-hari seperti pengelolaan data klien, pembuatan dan perubahan *invoice*, pencatatan pembayaran, dan peninjauan laporan keuangan. Berdasarkan hasil yang didapat, seluruh fitur utama pada sistem dapat berfungsi dengan baik dan sesuai dengan kebutuhan yang telah ditentukan. Baik *staff* keuangan maupun manajemen mengatakan bahwa sistem mampu menampilkan informasi keuangan secara jelas dan mudah dipahami. Secara keseluruhan, hasil pengetesan menunjukkan bahwa *website invoicing* telah berjalan sesuai dengan harapan pengguna dan siap digunakan untuk mendukung proses pencatatan serta pengelolaan keuangan perusahaan.

3.7 Kendala dan Solusi yang Ditemukan

3.7.1 Kendala

Selama masa pelaksanaan magang, terdapat beberapa kendala yang dihadapi dalam proses pengembangan sistem. Kendala-kendala tersebut diuraikan sebagai berikut:

- **Manajemen Waktu**: Terdapat kesulitan dalam mengatur dan membagi waktu secara efektif. Terutama ketika harus mempelajari hal-hal baru seperti teknologi atau *framework* yang belum familiar sebelumnya. Hal ini menyebabkan beberapa proses pembelajaran dan pengembangan berlangsung lebih lama dari jadwal yang direncanakan. Selain itu, hal lain yang menghambat adalah ditemukan banyak sekali *bug* saat pengembangan sehingga *project* selesai melebihi waktu yang diperkirakan.

- **Perubahan *Project***: Pada pertengahan masa magang, terjadi perubahan *project* dari pengembangan aplikasi *mobile* dengan menggunakan *framework* Flutter menjadi pembuatan sistem *invoicing* berbasis *website*. Perubahan ini menyebabkan adanya penyesuaian ulang terhadap *tech stack*, alur kerja, serta target implementasi yang telah direncanakan sebelumnya.
- **Kurangnya Komunikasi antara *Front-end* dan *Back-end***: Terjadi miskomunikasi dalam perencanaan dan implementasi API antara tim *front-end* dan *back-end*. Hal ini dikarenakan kurangnya dokumentasi API yang jelas serta absennya kesepakatan terkait struktur respons dan validasi input. Akibatnya, proses integrasi menjadi terhambat karena terjadi perbedaan terhadap data yang dikirim maupun diterima.

3.7.2 Solusi

- **Manajemen Waktu**: Untuk mengatasi permasalahan dalam manajemen waktu, hal yang bisa dilakukan adalah melakukan perencanaan mingguan yang lebih terstruktur menggunakan *task board* seperti Trello atau Microsoft Excel. Setiap pekerjaan dibagi ke dalam tugas-tugas kecil dengan tenggat waktu yang jelas. Selain itu, waktu pembelajaran untuk teknologi baru dijadwalkan secara paralel dengan pengerjaan tugas ringan, sehingga tidak terjadi penundaan total dalam proses pengembangan.
- **Perubahan *Project***: Tim segera melakukan penyesuaian dengan mengadakan diskusi untuk mendefinisikan ulang kebutuhan proyek baru. Riset teknologi pengganti dilakukan secara cepat dan intensif, serta disepakati penggunaan teknologi yang paling sesuai untuk pengembangan *website* dan kebutuhan dari sistem. Dengan adanya perencanaan ulang dan pembagian tugas yang tepat, transisi dari proyek sebelumnya ke proyek baru dapat dijalankan dengan lancar tanpa mengorbankan kualitas hasil.
- **Kurangnya Komunikasi antara *Front-end* dan *Back-end***: Untuk memperbaiki kendala komunikasi, cara yang diterapkan adalah penggunaan dokumentasi API dengan Swagger agar seluruh *endpoint* terdokumentasi dengan jelas. Selain itu, dilakukan pertemuan singkat rutin tiap harinya untuk menyinkronkan progres, membahas *error* yang muncul saat integrasi, serta memastikan kesesuaian antara data dari *front-end* dan *back-end*. Langkah ini berhasil mengurangi kesalahan dan mempercepat proses integrasi sistem.