BAB 3
PELAKSANAAN KERJA MAGANG

3.1 Kedudukan dan Koordinasi

Posisi selama pelaksanaan magang adalah sebagai *Backend Developer Intern* pada bagian IT *Developer* yang berada di bawah departemen Teknologi Informasi. Selama pelaksanaan magang, saya didampingi oleh mentor yang menjabat sebagai *Head of IT Department*, yang secara aktif memberikan arahan serta masukan selama proses pengembangan sistem *backend*. Koordinasi dilakukan melalui rapat rutin dan diskusi langsung untuk menampilkan progres serta mendapatkan *feedback* secara langsung. Kolaborasi antar *developer* difasilitasi menggunakan GitHub untuk mempermudah proses pengembangan dan pemisahan kode antar fitur. Selain itu, rapat mingguan bersama *Head of IT* juga dilakukan untuk membahas progres proyek serta kendala teknis yang dihadapi selama pengembangan.

3.2 Tugas yang Dilakukan

Selama pelaksanaan program kerja magang, tugas yang diberikan adalah mengembangkan *website* pembuatan *invoice*. *Website* ini memungkinkan divisi keuangan membuat *invoice* secara mandiri serta memudahkan pengubahan data apabila terjadi rekonsiliasi akibat perbedaan data dengan *database* atau pihak klien.

3.3 Uraian Pelaksanaan Magang

Tabel 3.1 merupakan daftar aktivitas yang dilakukan setiap minggunya selama pelaksanaan magang.

UNIVERSITAS MULTIMEDIA NUSANTARA

Tabel 3.1. Daftar aktivitas mingguan selama magang

Minggu	Aktivitas
1	Melakukan perkenalan dengan tim proyek, dilanjutkan dengan sesi *brainstorming* untuk merumuskan kebutuhan pengembangan aplikasi. Setelah itu, melakukan pemilihan tech stack yang akan digunakan, serta mempelajari bahasa pemrograman Go dan *framework* Gin sebagai bagian dari persiapan teknis proyek.
2	Mempelajari bahasa pemrograman Go dengan membuat proyek mini berupa *URL shortener* sebagai media latihan untuk memahami konsep dasar bahasa, serta penerapan *routing* dan *handler* pada *framework* Gin.
3	Mempelajari bahasa Go dan *framework* Gin melalui pembuatan proyek *to-do list* sebagai sarana memahami struktur dasar aplikasi, implementasi *routing*, serta pengelolaan data menggunakan metode *RESTful*.
4	Mempelajari bahasa Go dan *framework* Gin dengan membuat *website* ramalan cuaca yang mengambil data dari API *open source*, guna memahami integrasi API eksternal, pengolahan data JSON, serta penyajian informasi melalui *endpoint* dinamis.
5	Membuat skema *database* sebagai dasar perancangan sistem, serta mengikuti rapat finalisasi desain untuk memastikan seluruh kebutuhan telah sesuai. Setelah itu, memulai proses *setup project* dan tahap awal pengembangan sesuai dengan spesifikasi yang telah disepakati.
6	Melakukan proses *debugging* untuk memastikan fungsi berjalan sesuai harapan, serta mengembangkan fitur CRUD untuk entitas *Location* sebagai bagian dari pengelolaan data lokasi dalam sistem.
7	Melakukan *debugging* untuk memastikan stabilitas sistem, menyusun dokumentasi teknis guna mempermudah pemahaman dan pengembangan lebih lanjut, serta mengembangkan fitur peminjaman barang sebagai bagian dari fungsionalitas utama aplikasi.
8	Melanjutkan pengembangan fitur peminjaman barang, melakukan perbaikan *bug* yang ditemukan selama proses pengujian, serta melengkapi dokumentasi teknis agar seluruh alur dan fungsionalitas sistem terdokumentasi dengan baik.
	Lanjut pada halaman berikutnya

Tabel 3.1 Daftar aktivitas mingguan selama magang (lanjutan)

Minggu	Aktivitas
9	Mempelajari teori dasar terkait teknologi *cloud*, serta memahami konsep kontainerisasi menggunakan Docker dan automasi *pipeline* CI/CD menggunakan Jenkins sebagai persiapan untuk penerapan dalam pengembangan dan *deployment* aplikasi.
10	Menerapkan konsep *Cloud Computing*, Docker, dan Jenkins dalam sebuah proyek *dummy* untuk menguji penerapan dan fungsionalitas teknologi tersebut, kemudian mengintegrasikannya ke dalam proyek asli untuk memastikan skala dan kinerja sistem dapat berjalan secara optimal.
11	Melakukan *review* terhadap progres yang telah dilakukan sejauh ini, kemudian memperbaiki *bug* yang terkait dengan integrasi *auto hosting*. Selain itu, melakukan perubahan pada struktur folder *ecosystem* untuk meningkatkan organisasi dan efisiensi pengelolaan proyek.
12	Melanjutkan pengembangan fitur CRUD untuk entitas *User* dan Vendor, serta melakukan perbaikan *bug* yang ditemukan selama proses pengujian guna memastikan kelancaran fungsi dan kinerja aplikasi.
13	Mengadakan rapat untuk membahas proyek baru, dilanjutkan dengan sesi *brainstorming* untuk mencari solusi atas tantangan yang dihadapi. Setelah itu, melakukan pemilihan *tech stack* yang akan digunakan dalam pengembangan proyek untuk memastikan kesesuaian dengan kebutuhan dan tujuan sistem.
14	Membuat *controller* untuk fitur *user register*, *login*, dan *user management* sebagai bagian dari pengelolaan autentikasi dan data pengguna dalam sistem.
15	Membuat *middleware*, mengembangkan *controller* untuk entitas *client*, serta melakukan integrasi dengan *frontend* guna memastikan konektivitas dan fungsionalitas antarmuka berjalan dengan baik.
16	Melanjutkan pembuatan *controller* untuk entitas *client* dan *invoice* sebagai bagian dari pengembangan fitur pengelolaan data klien dan tagihan dalam sistem.

Lanjut pada halaman berikutnya

Tabel 3.1 Daftar aktivitas mingguan selama magang (lanjutan)

Minggu	Aktivitas
17	Membuat dokumentasi untuk entitas *user*, *invoice*, dan *client* guna memastikan seluruh alur dan fungsionalitas terkait terdokumentasi dengan jelas dan dapat dipahami untuk pengembangan lebih lanjut.
18	Mengimplementasikan Docker dan Jenkins untuk pengujian *pipeline* CI/CD guna memastikan proses *build*, *test*, dan *deployment* berjalan otomatis dan efisien.
19	Memperbaiki *bug* pada Jenkins yang menyebabkan kesulitan integrasi dengan GitHub *webhook*, guna memastikan proses otomatisasi *pipeline* CI/CD berjalan dengan lancar.
20	Melakukan integrasi sistem dengan *frontend* melalui penyelarasan *endpoint* yang telah dikembangkan, serta memberikan dukungan kepada tim *frontend* dalam memahami struktur *request* dan *response* dari sisi *backend* untuk memastikan integrasi berjalan dengan lancar.

3.4 Analisis dan perancangan sistem

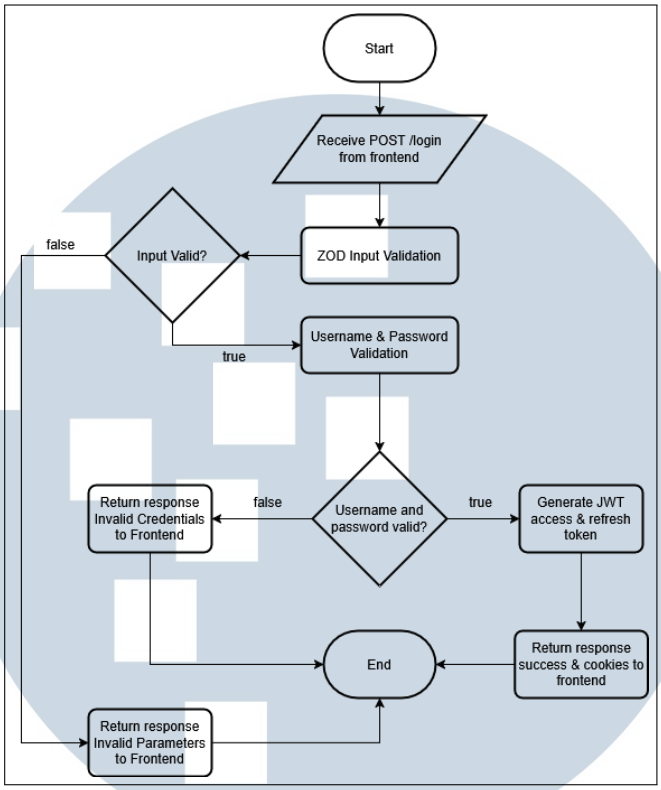
Uraian berikut menjelaskan tentang Pembuatan Sistem *Website Invoice* pada PT Mobile Data Indonesia. Sistem ini dirancang untuk mempermudah proses pembuatan dan pengelolaan *invoice* secara digital dalam lingkungan perusahaan. Dokumentasi yang disajikan mencakup diagram *Flowchart*, serta *Database Schema* untuk menggambarkan fungsionalitas utama, struktur basis data, dan alur proses sistem secara menyeluruh.

3.4.1 Flowchart

Rancangan *flowchart* untuk *endpoint backend* diuraikan dalam beberapa bagian berikut, yang masing-masing merepresentasikan alur proses dari fitur-fitur utama sistem. Penjelasan dimulai dari alur autentikasi pengguna, sebagaimana dijelaskan pada subbagian berikut:

A Flowchart: Login User

Untuk menggambarkan proses autentikasi pengguna, ditampilkan Diagram 3.1 yang memvisualisasikan rancangan alur login.



Gambar 3.1. Flowchart - login

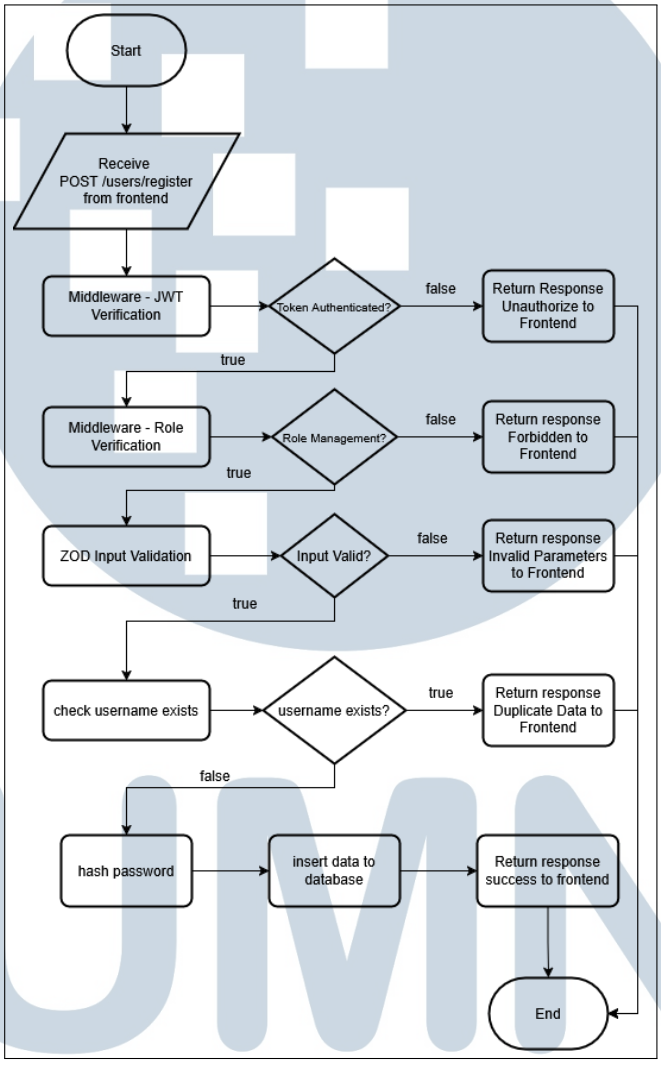
Ketika permintaan dikirimkan dari sisi klien, sistem terlebih dahulu akan memvalidasi data masukan (*input*) untuk memastikan bahwa seluruh data yang diperlukan telah disediakan dan sesuai dengan skema model yang ditetapkan. Jika data tidak valid, maka sistem akan segera mengembalikan respons `400 Invalid Parameters`.

Apabila *input* valid, sistem akan melakukan pengecekan kredensial dengan mencocokkan nama pengguna dan kata sandi terhadap data yang ada di basis data. Jika kredensial tidak sesuai, sistem akan memberikan respons `401 Invalid Credentials`.

Sebaliknya, jika kredensial dinyatakan *valid*, maka sistem akan menghasilkan dua buah token autentikasi: `access token` dan `refresh token`. Kedua token ini kemudian dikirimkan kembali ke klien dalam bentuk *cookies*, bersamaan dengan respons `200 Success`, sebagai tanda bahwa proses autentikasi berhasil.

B Flowchart: Register New User

Gambar 3.2 menunjukkan alur proses pendaftaran pengguna baru melalui endpoint `POST /users/register`.



Gambar 3.2. Flowchart - Register

Sebelum permintaan diproses, sistem akan menerapkan dua lapisan *middleware* pengamanan. Lapisan pertama adalah `JWT Verification`, yang memastikan token autentikasi tersedia, dapat di-*decode*, dan belum kedaluwarsa. Jika token tidak valid, sistem akan langsung mengembalikan respons `401 Unauthorized`.

Jika token valid, permintaan diteruskan ke `Role Verification`. Hanya pengguna dengan peran *management* yang diperbolehkan untuk membuat akun

baru. Permintaan dari peran lain akan ditolak dengan respons `403 Forbidden`.

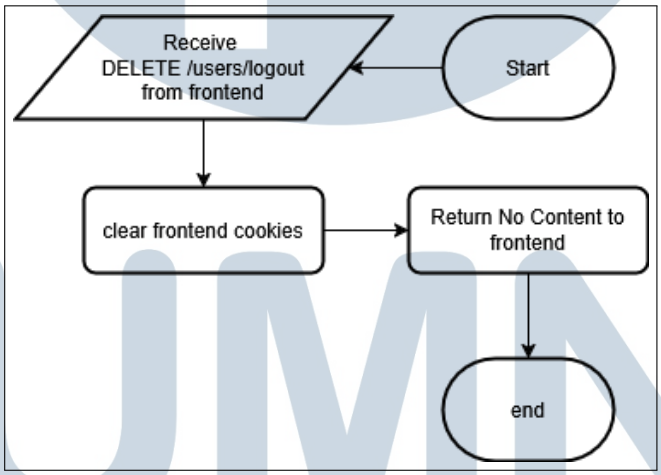
Setelah lolos otentikasi dan otorisasi, sistem akan melakukan validasi data masukan menggunakan `Zod`. Apabila data yang dikirim tidak sesuai dengan skema yang ditetapkan, sistem akan mengembalikan respons `400 Invalid Parameters`.

Jika data valid, sistem akan memeriksa apakah nama pengguna sudah terdaftar dalam basis data. Jika ditemukan duplikasi, maka akan dikembalikan respons `409 Duplicate Data`. Sebaliknya, jika tidak ada duplikasi, kata sandi akan di-*hash* dan seluruh data pengguna akan disimpan.

Sebagai penutup, sistem akan mengembalikan respons `201 Success` sebagai tanda bahwa akun baru berhasil dibuat.

C Flowchart: Log Out User

Gambar 3.3 menyajikan alur proses penghentian sesi pengguna melalui endpoint `DELETE /users/logout`.



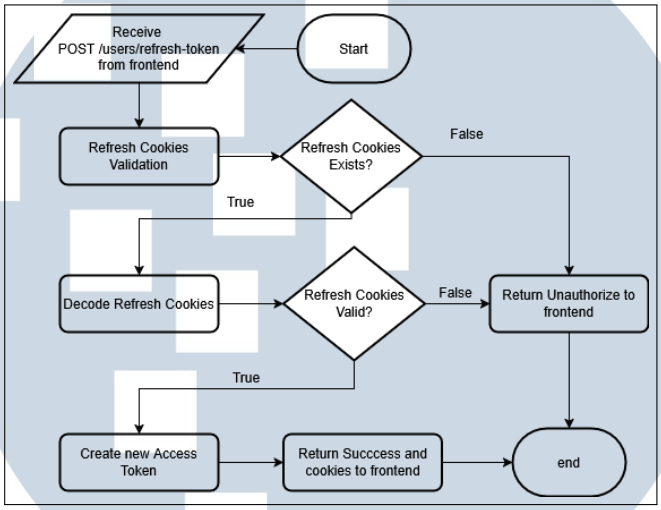
Gambar 3.3. Flowchart - Logout

Setelah menerima permintaan dari sisi klien, sistem akan menghapus *cookie* yang berisi token autentikasi. Tindakan ini menandakan bahwa sesi login pengguna telah dinonaktifkan.

Apabila proses penghapusan berhasil, sistem akan mengembalikan respons dengan status `204 No Content`, yang menunjukkan bahwa permintaan berhasil diproses tanpa ada data tambahan yang dikembalikan ke klien.

D Flowchart: Refresh Token

Gambar 3.4 menyajikan alur proses pembaruan token akses pengguna melalui endpoint `POST /users/refresh-token`.



Gambar 3.4. Flowchart - Refresh Token

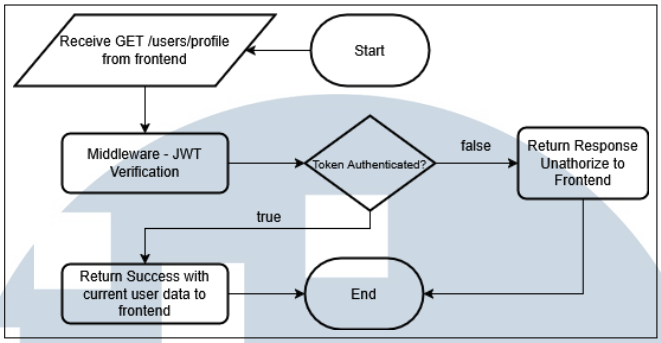
Setelah menerima permintaan dari sisi klien, sistem terlebih dahulu memeriksa apakah *refresh* token tersedia dalam *cookies* yang dikirimkan. Jika token tidak ditemukan, maka sistem akan langsung mengembalikan respons `401 Unauthorized`.

Jika *refresh* token tersedia, sistem akan mencoba melakukan proses *decode* terhadap token tersebut. Apabila proses *decoding* gagal atau token tidak valid, maka sistem akan memberikan respons `401 Unauthorized`.

Namun, apabila token berhasil di-*decode* dan dinyatakan valid, sistem akan menghasilkan *access* token baru dan mengembalikannya kepada klien dalam bentuk *cookies* sebagai bentuk autentikasi yang telah diperbarui. Respons yang diberikan adalah `200 Success`, yang menandakan bahwa proses pembaruan token berhasil dilakukan.

E Flowchart: Get Profile

Gambar 3.5 menggambarkan alur permintaan data profil pengguna yang sedang aktif melalui *endpoint* `GET /users/profile`.



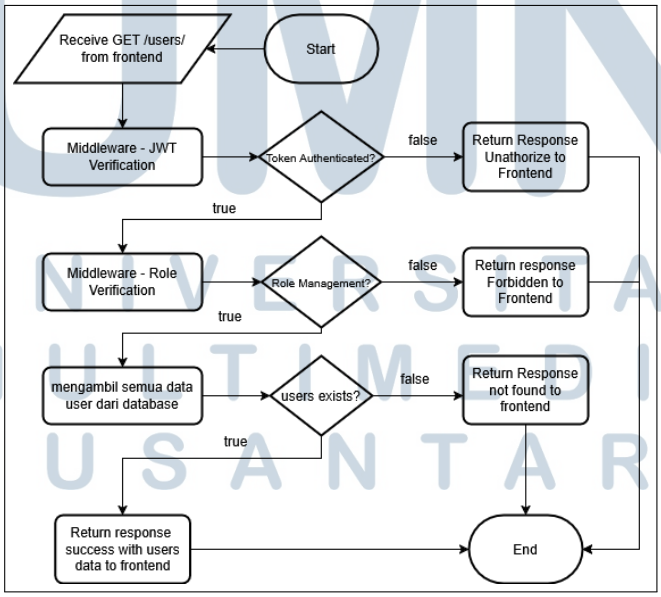
Gambar 3.5. Flowchart - Get Profile

Setelah permintaan diterima, sistem menjalankan proses verifikasi melalui `JWT verification`. *Middleware* ini bertugas untuk memastikan bahwa token autentikasi yang dikirimkan oleh klien tersedia, valid, dan belum kedaluwarsa. Jika token tidak tersedia atau tidak dapat di-*decode*, maka sistem akan mengembalikan respons `401 Unauthorized` ke sisi klien.

Jika token valid, sistem akan mengambil data pengguna aktif berdasarkan informasi yang terdapat di dalam token, lalu mengembalikan respons `200 Success` beserta data lengkap profil pengguna tersebut.

F Flowchart: Get All User

Gambar 3.6 menyajikan alur proses pengambilan seluruh data pengguna melalui *endpoint* `GET /users`.



Gambar 3.6. Flowchart - Get All User

Sebelum permintaan diproses lebih lanjut, sistem akan menjalankan dua lapisan *middleware* sebagai langkah pengamanan.

Pertama, sistem memverifikasi permintaan menggunakan `JWT verification` untuk memastikan bahwa token autentikasi tersedia, masih berlaku, dan dapat di-*decode* dengan benar. Jika token tidak tersedia, telah kedaluwarsa, atau gagal di-*decode*, maka sistem akan memberikan respons `401 Unauthorized`.

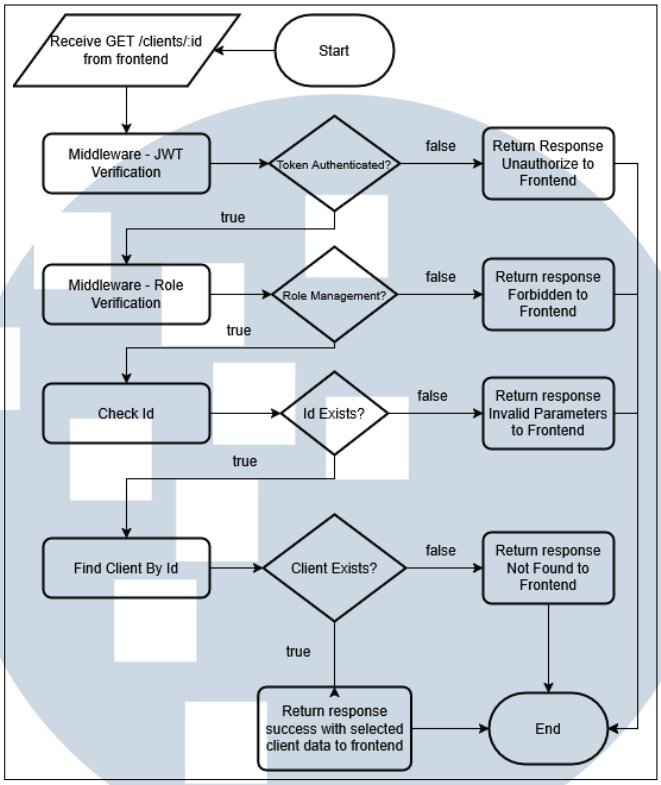
Jika token dinyatakan valid, permintaan dilanjutkan ke lapisan kedua, yaitu *Role verification*. Pada tahap ini, sistem memeriksa apakah pengguna memiliki peran *management*. Hanya pengguna dengan peran tersebut yang diizinkan mengakses *endpoint* ini. Jika peran tidak sesuai, maka sistem akan membalas dengan respons `403 Forbidden`.

Setelah lolos dari dua lapisan *middleware*, sistem akan menjalankan proses pengambilan data dari basis data. Jika tidak ada data pengguna ditemukan, maka akan dikembalikan respons `404 Not Found`. Namun, apabila data ditemukan, maka sistem akan mengembalikan respons `200 Success` beserta seluruh data pengguna kepada klien.

G Flowchart: Get User By ID

Gambar 3.7 memperlihatkan alur proses pengambilan data pengguna secara spesifik berdasarkan `id` melalui *endpoint* `GET /users/:id`.

UMN
UNIVERSITAS
MULTIMEDIA
NUSANTARA



Gambar 3.7. Flowchart - Get User By Id

Setelah menerima permintaan, sistem terlebih dahulu menjalankan *middleware* `JWT verification` untuk memastikan token autentikasi tersedia, valid, dan belum kedaluwarsa. Apabila token tidak ditemukan atau tidak dapat di-*decode*, maka sistem akan langsung mengembalikan respons `401 Unauthorized`.

Jika token valid, permintaan akan diteruskan ke *middleware* `Role verification` untuk memeriksa apakah pengguna memiliki peran *management*. Hanya peran tersebut yang diizinkan mengakses *endpoint* ini. Jika pengguna memiliki peran yang tidak sesuai, maka sistem akan memberikan respons `403 Forbidden`.

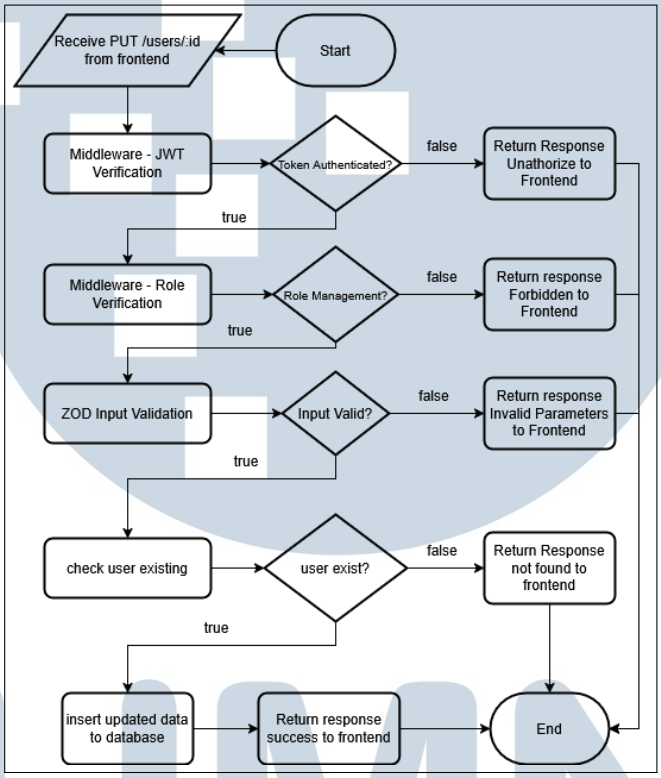
Setelah lolos dari dua lapisan *middleware*, permintaan akan diteruskan ke *controller*. Langkah pertama dalam *controller* adalah memvalidasi parameter `id`. Jika parameter tidak ada atau tidak valid, maka sistem akan mengembalikan respons `400 Invalid ID`.

Jika parameter valid, sistem akan melakukan pencarian data *user* berdasarkan `id` di dalam *database*. Apabila data tidak ditemukan, sistem akan memberikan respons `404 Not Found`. Namun, jika data ditemukan, maka sistem akan mengembalikan respons `200 Success` disertai dengan data lengkap *user* yang

dimaksud.

H Flowchart: Update User

Gambar 3.8 menyajikan alur proses pembaruan data pengguna melalui *endpoint* `PUT /users/:id`.



Gambar 3.8. Flowchart - Update User

Setelah menerima permintaan, proses pertama yang dijalankan adalah *middleware* `JWT verification`, yang bertugas untuk memverifikasi keberadaan dan validitas token. Jika token tidak ada atau tidak dapat di-*decode*, maka sistem akan langsung mengembalikan respons `401 Unauthorized` ke klien.

Apabila token valid, permintaan akan diteruskan ke bagian *controller*. Di dalam *controller* ini, dilakukan validasi data masukan menggunakan Zod untuk memastikan bahwa data yang dikirim sesuai dengan format dan aturan yang ditetapkan. Jika data tidak valid, sistem akan mengirimkan respons `400 Invalid Parameters`.

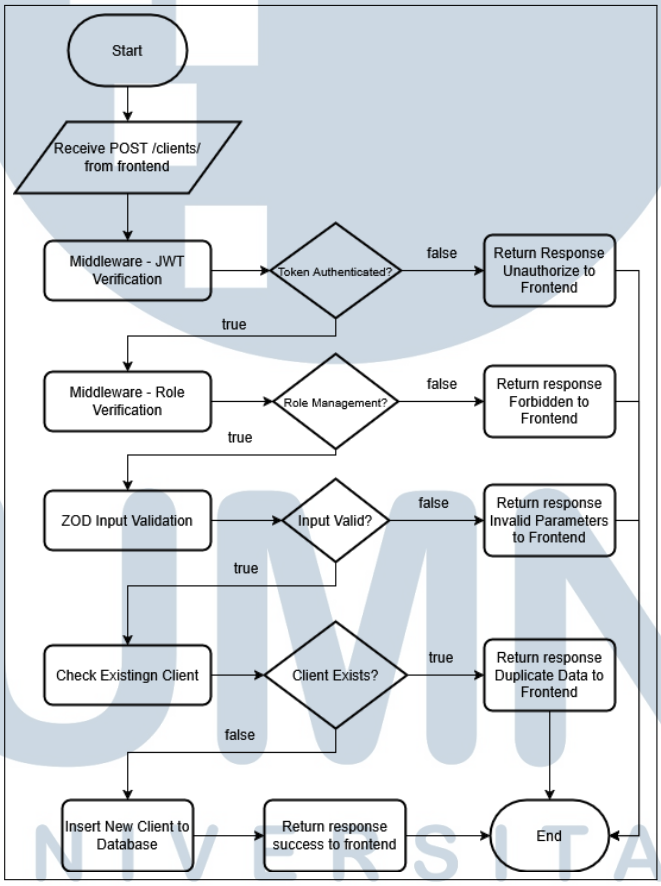
Selanjutnya, apabila data masukan valid, sistem akan memeriksa apakah *user* dengan `id` yang diberikan benar-benar ada di dalam *database*. Jika tidak

ditemukan, maka akan dikembalikan respons `404 Not Found`. Namun, jika user ditemukan, maka proses pembaruan data akan dilakukan, dan sistem akan menyimpan data terbaru ke dalam *database*.

Setelah proses berhasil, sistem akan memberikan respons `200 Success` sebagai konfirmasi bahwa data pengguna berhasil diperbarui.

I Flowchart: Create New Client

Gambar 3.9 menyajikan alur proses penambahan data klien baru melalui *endpoint* `POST /clients`.



Gambar 3.9. Flowchart - Create Client

Sebelum permintaan diproses lebih lanjut, sistem akan menjalankan dua lapisan *middleware*. Lapisan pertama adalah `JWT Verification`, yang bertujuan untuk memastikan bahwa hanya pengguna yang telah terautentikasi yang dapat mengakses *endpoint* ini. Jika token tidak tersedia atau gagal di-*decode*, maka sistem akan mengembalikan respons `401 Unauthorized`.

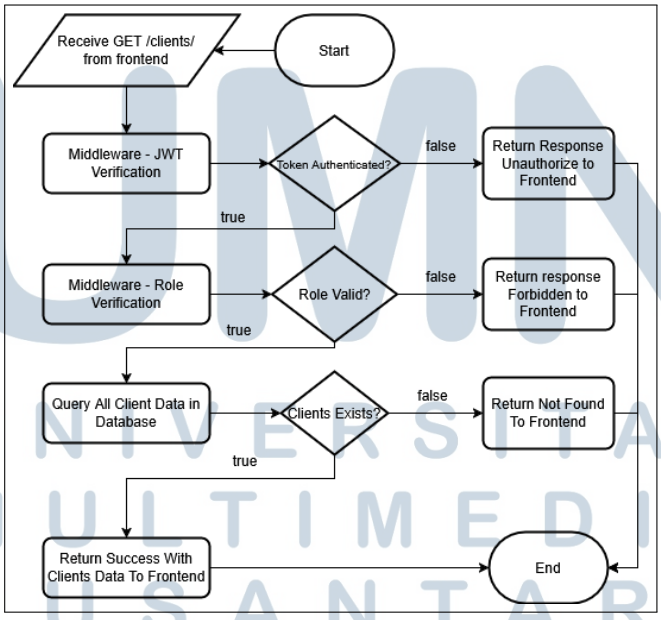
Jika token valid, permintaan dilanjutkan ke lapisan kedua yaitu `Role Verification`. Pada *endpoint* ini, hanya pengguna dengan peran *finance* yang memiliki izin untuk mengakses. Jika peran tidak sesuai, maka sistem akan memberikan respons `403 Forbidden`.

Setelah melewati proses autentikasi dan otorisasi, sistem akan melanjutkan ke bagian *controller*. Di tahap ini, sistem akan melakukan validasi terhadap *input* menggunakan pustaka Zod, guna memastikan bahwa data yang dikirim sesuai dengan struktur dan ketentuan basis data. Jika validasi gagal, maka akan dikembalikan respons `400 Invalid Parameters`.

Jika data valid, sistem akan melakukan pemeriksaan apakah klien dengan data yang sama sudah ada di basis data. Jika klien telah terdaftar sebelumnya, sistem akan mengembalikan respons `409 Duplicate Data`. Namun, jika belum ada, maka data klien akan dimasukkan ke dalam basis data, dan sistem akan memberikan respons `201 Success` sebagai tanda bahwa data berhasil ditambahkan.

J Flowchart: Get All Client

Gambar 3.10 menyajikan alur proses pengambilan seluruh data klien melalui *endpoint* `GET /clients`.



Gambar 3.10. Flowchart - Get All Clients

Proses eksekusi dimulai dengan validasi autentikasi menggunakan `JWT verification`. Jika token tidak tersedia, gagal di-decode, atau sudah tidak valid,

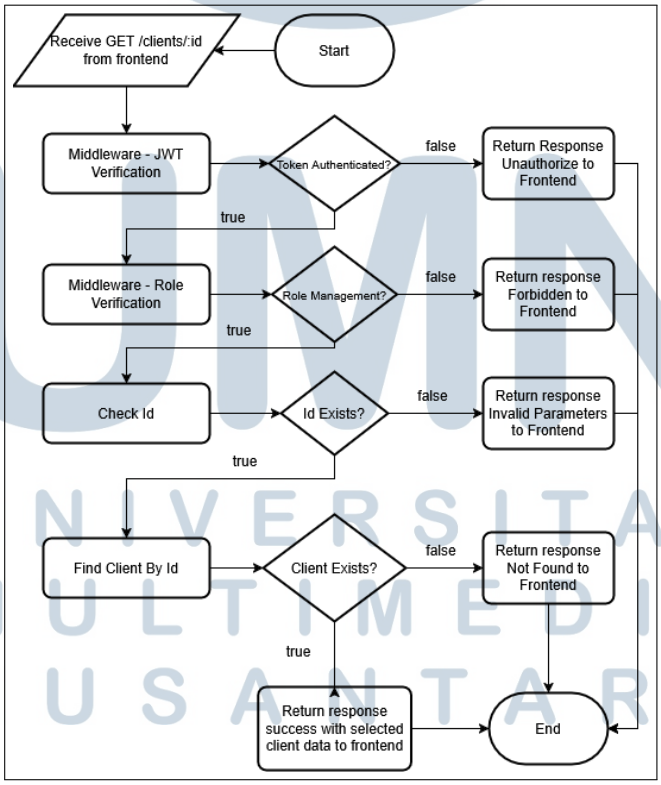
maka sistem akan mengembalikan respons `401 Unauthorized`.

Jika token berhasil divalidasi, permintaan akan diteruskan ke lapisan *middleware* berikutnya, yaitu `role verification`, untuk memverifikasi apakah pengguna memiliki peran yang sesuai. Hanya pengguna dengan peran `finance` atau `management` yang diperbolehkan mengakses *endpoint* ini. Apabila peran pengguna tidak sesuai, maka akan diberikan respons `403 Forbidden`.

Setelah proses autentikasi dan otorisasi berhasil, sistem akan meneruskan permintaan ke bagian *controller*, di mana proses query akan dijalankan untuk mengambil semua data *client* dari *database*. Jika tidak ada data yang ditemukan, maka sistem akan mengembalikan respons `404 Not Found`. Namun, apabila data tersedia, sistem akan memberikan respons `200 Success` beserta seluruh data *client* kepada sisi *frontend*.

K Flowchart: Get Client By Id

Gambar 3.11 menyajikan alur proses pengambilan data klien secara spesifik berdasarkan `id` melalui *endpoint* `GET /clients/:id`.



Gambar 3.11. Flowchart - Get Client By Id

Sebelum proses dilanjutkan, sistem terlebih dahulu menerapkan middleware `JWT verification` untuk memastikan bahwa pengguna yang melakukan permintaan sudah terautentikasi. Jika token tidak tersedia atau tidak dapat di-*decode*, maka sistem akan mengembalikan respons `401 Unauthorized`.

Apabila token valid, permintaan akan diteruskan ke *middleware* `role verification` untuk memverifikasi hak akses pengguna. Hanya pengguna dengan peran `management` dan `finance` yang diizinkan mengakses *endpoint* ini. Jika pengguna memiliki peran di luar kedua *role* tersebut, sistem akan mengembalikan respons `403 Forbidden`.

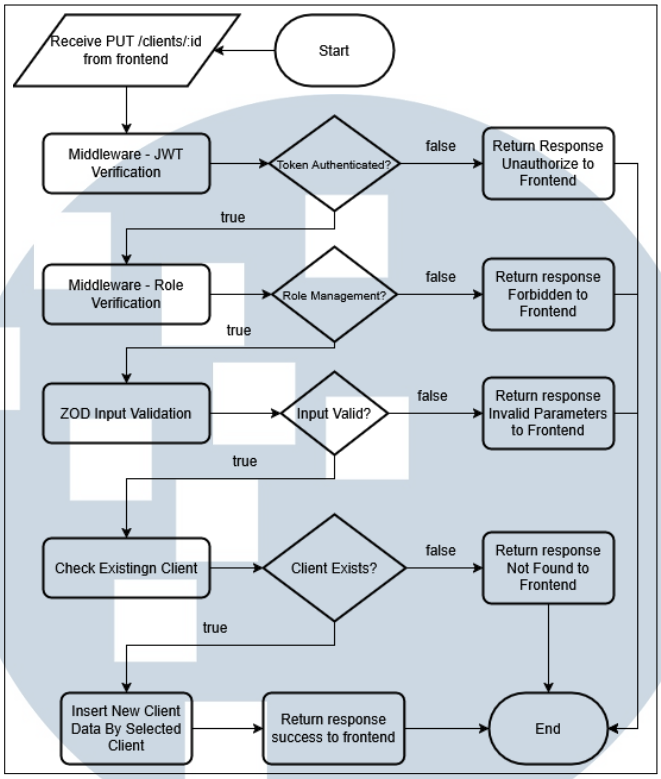
Selanjutnya, sistem akan memvalidasi parameter `id` yang dikirimkan melalui URL. Jika parameter `id` tidak ditemukan atau tidak valid, maka sistem akan merespons dengan `400 Invalid Parameters`.

Apabila parameter valid, sistem akan melakukan pencarian data client berdasarkan `id` tersebut di dalam *database*. Jika data *client* tidak ditemukan, maka sistem akan memberikan respons `404 Not Found`. Namun, jika *client* ditemukan, sistem akan mengembalikan data lengkap dengan respons `200 Success`.

L Flowchart: Update Client By Id

Gambar 3.12 memperlihatkan alur proses pembaruan data klien yang telah terdaftar melalui *endpoint* `PUT /clients/:id`, termasuk penerapan dua lapisan *middleware* pengaman sebelum permintaan diproses oleh sistem.

UMN
UNIVERSITAS
MULTIMEDIA
NUSANTARA



Gambar 3.12. Flowchart - Update Client

Lapisan pertama adalah `JWT verification`, yang memvalidasi keberadaan serta keabsahan token autentikasi. Jika token tidak tersedia, gagal diuraikan, atau telah kedaluwarsa, sistem akan mengembalikan respons `401 Unauthorized`.

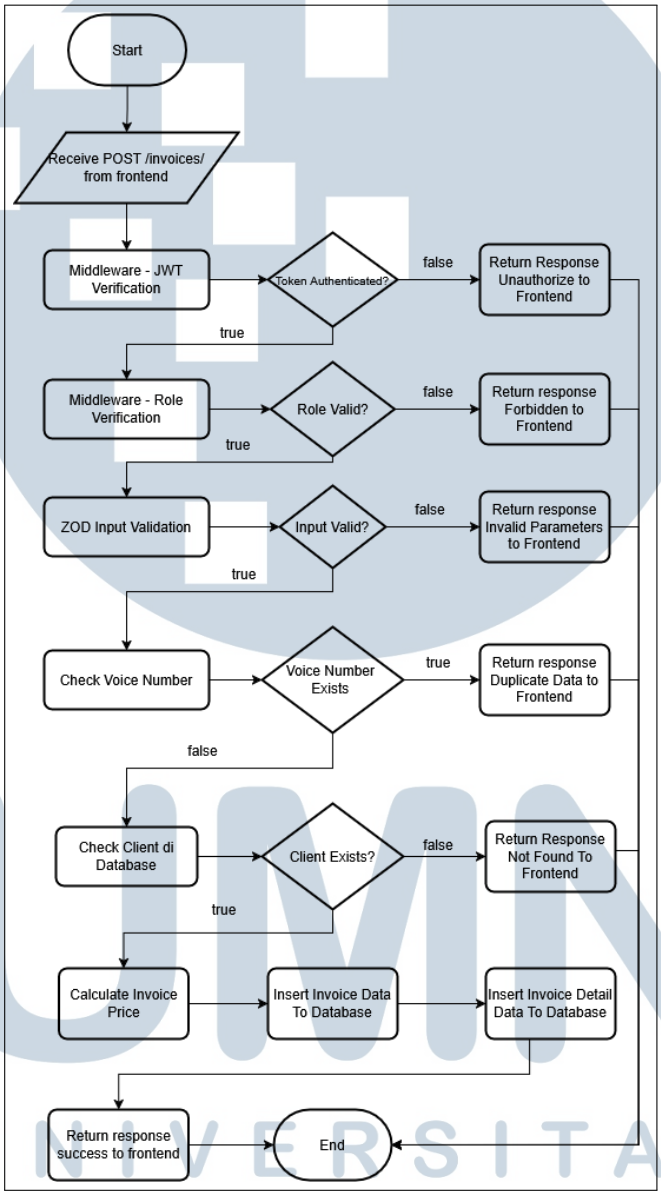
Jika token valid, permintaan dilanjutkan ke `role verification`. Pada endpoint ini, hanya pengguna dengan peran *finance* yang diizinkan melakukan pembaruan data *client*. Upaya akses oleh peran lain akan menghasilkan respons `403 Forbidden`.

Setelah berhasil melewati kedua lapisan tersebut, sistem mengeksekusi validasi masukan menggunakan Zod untuk memastikan data yang dikirim sesuai dengan skema basis data. Kegagalan validasi diproses dengan respons `400 Invalid Parameters`.

Jika data valid, *controller* akan memeriksa keberadaan *client* berdasarkan `id`. Bila *client* tidak ditemukan, sistem akan mengembalikan respons `404 Not Found`. Sebaliknya, jika *client* ditemukan, sistem akan memperbarui data sesuai masukan dan mengembalikan respons `200 Success` sebagai konfirmasi bahwa pembaruan berhasil dilakukan.

M Flowchart: Create New Invoice

Gambar 3.13 menggambarkan alur proses pembuatan *invoice* baru melalui *endpoint* terkait, dimulai dari tahap validasi token JWT oleh *middleware*.



Gambar 3.13. Flowchart - Create Invoice

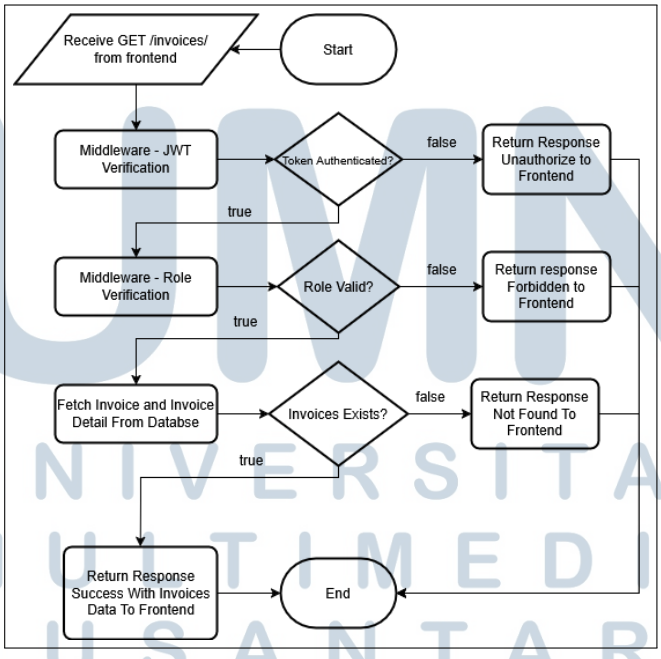
Tahap pertama apabila token tidak valid atau gagal didekode, sistem akan mengembalikan respons `401 Unauthorized`. Jika validasi token berhasil, proses dilanjutkan dengan pengecekan *role* melalui *middleware* verifikasi *role*. Endpoint ini hanya dapat diakses oleh pengguna dengan peran *finance*. Apabila peran pengguna tidak sesuai, sistem akan memberikan respons `403 Forbidden`.

Selanjutnya, input yang dikirimkan akan divalidasi menggunakan Zod untuk memastikan data sesuai dengan skema yang telah ditentukan. Jika data tidak valid, sistem akan memberikan respons `400 Invalid Parameters`. Jika validasi input berhasil, sistem akan mengecek apakah nomor *invoice* yang dikirimkan sudah pernah digunakan. Jika nomor tersebut sudah ada di *database*, sistem akan mengembalikan respons `409 Duplicate Data`.

Jika nomor *invoice* belum pernah digunakan, sistem akan memverifikasi keberadaan client berdasarkan data yang dikirim. Bila *client* tidak ditemukan, respons `404 Not Found` akan diberikan. Jika ditemukan, proses dilanjutkan ke tahap perhitungan harga *invoice*. Setelah semua data siap, sistem akan menyimpan *invoice* dan detailnya ke dalam *database* menggunakan transaksi untuk menjaga integritas data. Jika penyimpanan berhasil, sistem akan mengembalikan respons `200 Success` ke *frontend*.

N Flowchart: Get All Invoice

Endpoint `GET /invoices` menyediakan daftar lengkap *invoice* sebagaimana diilustrasikan pada Gambar 3.14.



Gambar 3.14. Flowchart - Get All Invoice

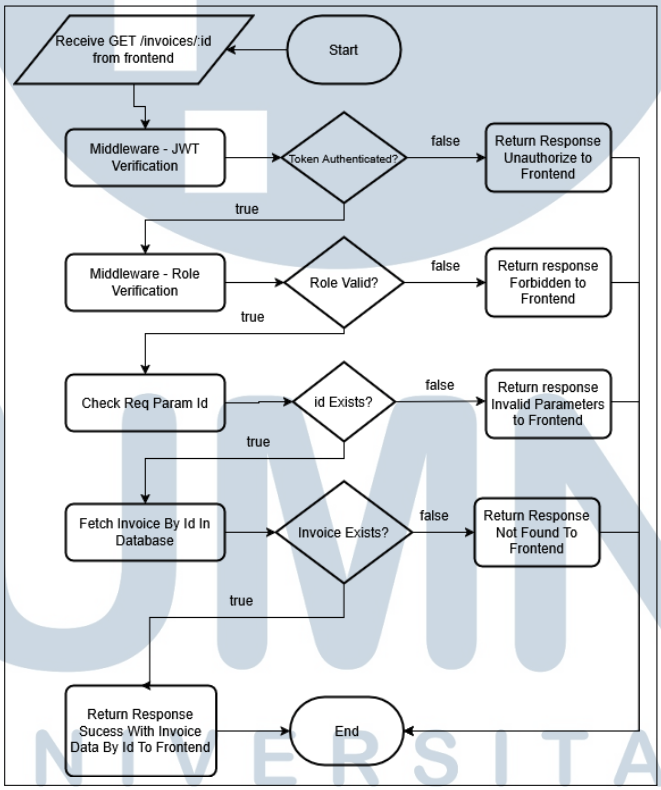
Akses ke *endpoint* ini dijaga oleh dua *middleware*. Pertama, `JWT verification` memvalidasi token sesi, apabila token tidak sah atau kosong

menghasilkan respons `401 Unauthorized`. Setelah lolos, permintaan diteruskan ke `role validation`, yang membatasi akses untuk peran *management* dan *finance*, peran lain menerima respons `403 Forbidden`.

Jika kedua lapis validasi berhasil, aplikasi mengeksekusi proses pengambilan data dari basis data. Apabila tidak ada data ditemukan, sistem membalas dengan `404 Not Found`. Sebaliknya, jika data tersedia, server mengembalikan respons `200 Success` berisi koleksi *invoice* yang diminta.

O Flowchart: Get Invoice By Id

Gambar 3.15 memperlihatkan alur proses pengambilan data *invoice* berdasarkan `id` melalui *endpoint* `GET /invoices/:id`.



Gambar 3.15. Flowchart - Get Invoice By Id

Endpoint `GET /invoices/:id` berfungsi untuk mengambil satu data *invoice* berdasarkan `id` yang diberikan pada *path* parameter, sebagaimana dijelaskan dalam Gambar 3.15. Seperti *endpoint* lainnya, proses diawali dengan `JWT verification` untuk memvalidasi token sesi. Jika token tidak valid atau kosong, maka sistem akan memberikan respons `401 Unauthorized`.

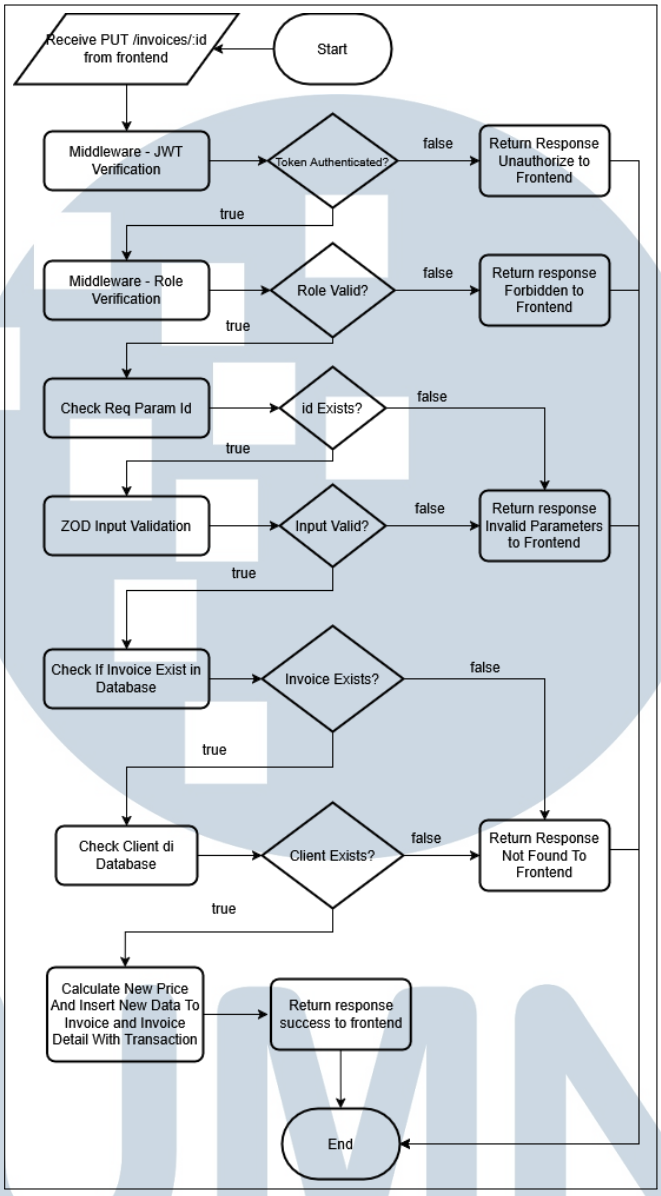
Apabila validasi token berhasil, permintaan dilanjutkan ke role validation, yang memastikan hanya pengguna dengan peran *management* atau *finance* yang dapat mengakses endpoint ini. Jika peran pengguna tidak sesuai, akan diberikan respons `403 Forbidden`.

Setelah lolos dari kedua lapisan *middleware*, sistem akan memeriksa apakah `id` terdapat pada *request* parameter. Jika tidak tersedia, maka sistem akan mengembalikan respons `400 Invalid Parameters`. Jika parameter tersedia, sistem melanjutkan untuk melakukan proses pengambilan data dari basis data berdasarkan `id` tersebut. Jika data tidak ditemukan, maka respons yang dikembalikan adalah `404 Not Found`. Sebaliknya, apabila data ditemukan, maka akan diberikan respons `200 Success` disertai dengan data *invoice*.

P Flowchart: Update Invoice By Id

Gambar 3.16 menyajikan alur proses pembaruan data *invoice* beserta detail terkait melalui *endpoint* `PUT /invoices/:id`.

UMN
UNIVERSITAS
MULTIMEDIA
NUSANTARA



Gambar 3.16. Flowchart - Update Invoice

Akses terhadap *endpoint* ini dilindungi oleh dua lapisan *middleware*. Pertama, proses `JWT verification` memastikan bahwa token sesi yang disertakan valid, apabila token tidak sah atau tidak tersedia, sistem akan membalas dengan status `401 Unauthorized`. Jika validasi token berhasil, permintaan dilanjutkan ke tahap `role validation`, yang memastikan hanya pengguna dengan peran *finance* yang dapat melanjutkan. Permintaan dari pengguna dengan peran lain akan ditolak dengan respons `403 Forbidden`.

Di sisi controller, sistem terlebih dahulu mengecek keberadaan parameter `id` pada *path*. Jika parameter tersebut tidak ditemukan, permintaan dianggap

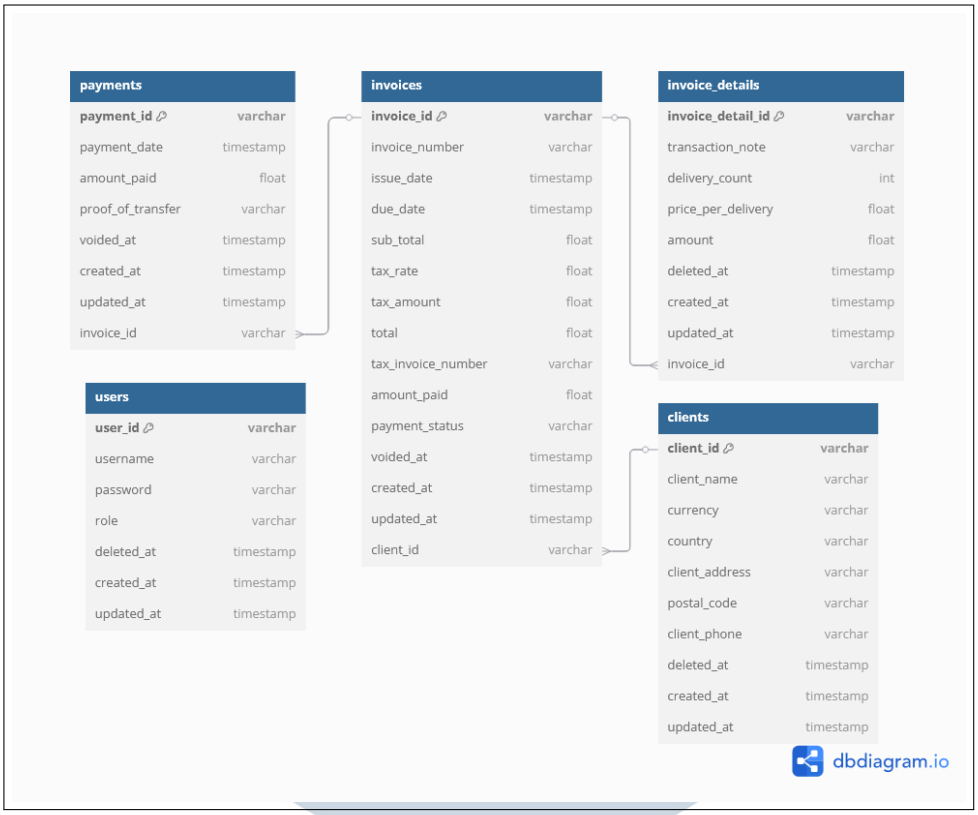
tidak valid dan dibalas dengan status `400 Invalid Parameters`. Setelah itu, isi permintaan diperiksa menggunakan validasi *input* Zod. Gagalnya validasi pada tahap ini juga akan memicu respons serupa.

Jika seluruh validasi berhasil dilewati, sistem melanjutkan dengan memverifikasi keberadaan *invoice* di basis data. Jika *invoice* yang dimaksud tidak tersedia, sistem akan merespons dengan `404 Not Found`. Pemeriksaan yang sama juga dilakukan terhadap entitas *client* yang terkait. Setelah kedua entitas dipastikan ada, proses pembaruan dimulai, termasuk perhitungan ulang terhadap harga. Seluruh proses ini dijalankan dalam satu transaksi untuk menjamin integritas dan konsistensi antara dua tabel yang saling terkait. Apabila seluruh langkah berhasil dilaksanakan tanpa kesalahan, sistem akan mengembalikan respons `200 Success` sebagai penanda keberhasilan operasi.

3.4.2 Database Schema

Gambar 3.17 menunjukkan skema basis data sistem *website invoice*, yang terdiri dari lima entitas utama: `users`, `clients`, `invoices`, `invoice_details`, dan `payments`. Diagram ini menggambarkan struktur data serta relasi antar tabel yang dirancang untuk mendukung proses pencatatan dan pengelolaan tagihan secara terintegrasi.

UMN
UNIVERSITAS
MULTIMEDIA
NUSANTARA



Gambar 3.17. Database Schema - Invoice

A Users

Menyimpan data pengguna, termasuk `username`, `password`, dan `role`. Atribut `deleted_at`, `created_at`, dan `updated_at` digunakan untuk kebutuhan audit.

B Clients

Berisi informasi klien penerima tagihan, seperti `client_name`, `currency`, `country`, serta kontak dan alamat. Satu klien dapat memiliki banyak *invoice*.

C Invoices

Mencatat faktur/tagihan, termasuk nomor *invoice*, tanggal terbit, jatuh tempo, subtotal, pajak, dan total pembayaran. Terhubung ke entitas klien melalui `client_id` sebagai *foreign key*.

D Invoice Details

Menampung rincian tiap *invoice*, seperti jumlah pengiriman, harga satuan, dan catatan transaksi. Satu *invoice* dapat memiliki banyak detail.

E Payments

Mencatat pembayaran atas suatu *invoice*, meliputi tanggal pembayaran, nominal, dan bukti transfer. Relasi ke *invoice* dilakukan melalui `invoice_id`.

Relasi antar tabel mencerminkan alur bisnis sistem: satu klien dapat memiliki banyak *invoice*, satu *invoice* dapat memiliki beberapa detail dan pembayaran, sehingga memungkinkan pencatatan transaksi yang fleksibel dan terdokumentasi dengan baik.

3.5 Implementasi

Dibagian ini akan dijelaskan mengenai hasil dari rancangan yang telah dilakukan sebelumnya. pada bagian implementasi ini akan lebih membahas mengenai Implementasi dari pembuatan *invoice*

3.5.1 Desain Rute

Dalam pengembangan aplikasi berbasis *web*, pengaturan rute (*routing*) menjadi komponen penting dalam mengarahkan permintaan klien (*client request*) ke fungsi pengendali (*controller function*) yang sesuai. Proses *routing* ini bertujuan untuk memisahkan logika antar fitur dalam sistem serta mempermudah pemeliharaan dan pengembangan di masa depan.

Pada implementasi sistem ini, digunakan pustaka Express sebagai kerangka kerja utama untuk *manajemen server* dan *routing*. kode 3.1 menunjukkan cuplikan kode dari file `app.ts`, yang berfungsi sebagai entry point dalam mengatur rute-rute utama aplikasi.

```
1 import express from 'express';
2 import invoiceRoute from '@routes/invoice.route';
3 //import lainnya
4 const app = express();
5
```

```
// Mengarahkan permintaan dengan prefix '/clients' ke modul routing client
app.use('/clients', clientRoute);

// Mengarahkan permintaan dengan prefix '/invoices' ke modul routing invoice
app.use('/invoices', invoiceRoute);

//routing lainnya
```

Kode 3.1: Cuplikan Kode - app.ts

Pada potongan kode tersebut, terlihat bahwa setiap *endpoint* utama didelegasikan ke file modul masing-masing melalui perintah `app.use`. Hal ini menandakan adanya arsitektur modular yang diterapkan, di mana setiap fitur memiliki berkas *routing* tersendiri yang menangani operasional secara spesifik. Strategi ini tidak hanya meningkatkan keterbacaan kode, tetapi juga mendukung prinsip pemisahan tanggung jawab.

Selanjutnya, setiap modul *routing* seperti `invoice.route.ts` berperan untuk mengelompokkan dan menangani rute yang lebih spesifik terhadap entitas terkait, dalam hal ini entitas *invoice*. Kode 3.2 menunjukkan pengelompokan rute yang dikelola oleh modul `invoice.route.ts`.

```
import { Router } from 'express';
//import route lainnya

//import middleware

const router = Router();

// Menangani permintaan POST untuk membuat invoice, hanya diakses oleh pengguna dengan peran 'finance'
router.post('/', authGuard, roleGuard(['finance']), createInvoiceController);

// Menangani permintaan GET untuk mengambil seluruh data invoice, diakses oleh peran 'finance' dan 'management'
router.get('/', authGuard, roleGuard(['finance', 'management']), getAllInvoiceController);

//route lainnya

```

```
16 export default router;
```

Kode 3.2: Cuplikan Kode - invoice.route.ts

Berdasarkan cuplikan kode tersebut, dapat diamati bahwa setiap rute dilindungi oleh *middleware* seperti `authGuard` dan `roleGuard`, yang bertugas memastikan bahwa hanya pengguna yang telah terautentikasi dan memiliki hak akses yang sesuai yang dapat mengakses rute tersebut. Misalnya, hanya pengguna dengan peran *finance* yang diperkenankan untuk membuat *invoice* melalui metode `POST`, sedangkan akses untuk membaca seluruh *invoice* terbuka bagi pengguna dengan peran *finance* maupun *management*.

Pendekatan ini tidak hanya menjamin keamanan aplikasi melalui kontrol akses berbasis peran, tetapi juga memberikan fleksibilitas dalam pengelolaan hak akses pada setiap rute secara modular dan terstruktur. Hal ini penting dalam konteks pengembangan sistem skala menengah hingga besar, di mana kontrol akses dan pemisahan logika bisnis menjadi krusial.

3.5.2 Middleware

Dalam arsitektur aplikasi berbasis *web*, khususnya pada pengembangan dengan kerangka kerja seperti Express, *middleware*merupakan komponen krusial yang berperan sebagai perantara antara permintaan (*request*) yang diterima dan pengendali utama (*controller*) yang akan menanganinya. *Middleware* memungkinkan aplikasi untuk melakukan serangkaian proses validasi atau transformasi data sebelum permintaan mencapai logika bisnis inti.

Sebelum sebuah permintaan diproses oleh *controller* yang relevan, sistem akan terlebih dahulu melewati satu atau beberapa middleware. *Middleware* ini berfungsi untuk memeriksa autentikasi pengguna, otorisasi peran, validasi data, *logging*, dan lain sebagainya. Dengan demikian, *middleware* berkontribusi dalam menjamin keamanan, efisiensi, dan konsistensi alur pemrosesan permintaan.

Berikut ini adalah beberapa *middleware* utama yang digunakan pada sebagian besar endpoint dalam sistem:

A Middleware Autentikasi

Middleware autentikasi bertugas untuk memastikan bahwa setiap permintaan yang masuk berasal dari pengguna yang telah berhasil melewati proses autentikasi. Mekanisme ini dilakukan dengan memverifikasi token akses

(*access token*) yang dikirimkan oleh klien melalui `cookie`. Cuplikan kode 3.3 berikut menunjukkan implementasi dari *middleware* autentikasi:

```
const accessToken = req.cookies['accessToken'];

if (!accessToken) {
    // return unauthorized
}

try {
    const decoded = jwt.verify(accessToken, /* secret */);

    const validate = await database.findUser({
    username: decoded.username,
    role: decoded.role,
    });

    if (!validate) {
        // return unauthorized
    }

    req.user = validate;

    next();
} catch (err) {
    // return unauthorized with error message
}
```

Kode 3.3: Cuplikan Kode - Middleware JWT

Pada cuplikan kode di atas, proses diawali dengan pengambilan token autentikasi dari objek `cookie` pada permintaan pengguna. Apabila token tidak ditemukan, maka permintaan langsung ditolak dengan respons tidak sah (`unauthorized`). Jika token tersedia, sistem akan mencoba memverifikasi keabsahan token menggunakan pustaka json web token (JWT).

Setelah token berhasil diverifikasi, sistem melanjutkan untuk mencocokkan identitas pengguna yang terenkapsulasi dalam token dengan data pengguna yang tersimpan di dalam basis data. Pencocokan ini dilakukan untuk memastikan bahwa pengguna benar-benar terdaftar dan memiliki peran yang sesuai sebagaimana tercantum dalam token. Jika validasi gagal, maka akses tetap akan ditolak. Namun jika berhasil, informasi pengguna tersebut akan disematkan ke objek permintaan (`req.user`), sehingga dapat digunakan oleh proses selanjutnya pada *controller*.

B Middleware Role

Middleware role-based authorization merupakan lapisan keamanan tambahan yang digunakan untuk memastikan bahwa hanya pengguna dengan peran tertentu yang dapat mengakses suatu *endpoint* dalam sistem. Setelah proses autentikasi berhasil dilakukan dan identitas pengguna dikenali, sistem perlu melakukan validasi terhadap hak akses berdasarkan peran pengguna tersebut.

Kode 3.4 Berikut merupakan *middleware* yang bertugas memverifikasi apakah peran pengguna telah sesuai dengan daftar peran yang diperbolehkan untuk mengakses suatu sumber daya:

```
1    try {
2        // Memeriksa apakah peran pengguna termasuk dalam daftar
     peran yang diizinkan
3        if (!allowedRoles.includes(req.user.role)) {
4            // return unauthorized
5        }
6
7        // Lanjut ke proses berikutnya jika validasi berhasil
8        next();
9    } catch (error) {
10       // Menangani kesalahan tak terduga
11       next(error);
12   }
```

Kode 3.4: Cuplikan Kode - Middleware Role

Dalam cuplikan kode di atas, logika pemeriksaan dilakukan dengan cara mencocokkan peran pengguna (`req.user.role`) terhadap daftar peran yang diizinkan (`allowedRoles`). Jika peran pengguna tidak termasuk dalam daftar tersebut, maka permintaan akan dihentikan, dan sistem akan memberikan respons tidak sah (`unauthorized`). Sebaliknya, jika validasi berhasil, alur permintaan akan diteruskan ke proses berikutnya, biasanya menuju *controller*.

Blok `try-catch` digunakan untuk menangani kemungkinan kesalahan yang terjadi selama proses validasi, seperti kesalahan sistem internal. Dengan demikian, *middleware* ini tidak hanya menjalankan fungsi otorisasi, tetapi juga memastikan stabilitas sistem dalam menghadapi skenario tak terduga.

3.5.3 Controller & Service

Pemisahan antara *controller* dan *service* bertujuan untuk menjembatani alur antara permintaan dari pengguna dan logika bisnis sistem. Dengan demikian, kode menjadi lebih terstruktur, modular, dan mudah dipelihara.

A Login Pengguna

Proses *authentication* atau otentikasi pengguna merupakan komponen yang sangat penting dalam sistem, karena memastikan bahwa hanya pengguna yang memiliki kredensial yang valid yang dapat mengakses fitur-fitur yang tersedia. Validasi kredensial dilakukan sebagai langkah awal sebelum sistem memberikan izin akses kepada pengguna. Dalam implementasinya, proses autentikasi ditangani melalui permintaan yang diterima oleh *controller*, yang kemudian diteruskan ke *service* untuk diproses lebih lanjut. Cuplikan kode 3.5 berikut menunjukkan bagaimana fungsi *controller* menangani permintaan login dari sisi pengguna.

```
export const loginController = async (req: Request, res: Response)
    : Promise<void> => {
    try {
        // Melakukan validasi terhadap data login dari body
        const validate = await userLoginSchema.safeParseAsync(req.
    body);

        if (!validate.success) {
            const parsed = parseZodError(validate.error);
            // mengembalikan response invalid parameter
            return;
        }

        // loginService akan mengembalikan token jika login
    berhasil
        const { accessToken, refreshToken } = await loginService(
    validate.data);

        const isProduction = process.env.NODE_ENV === 'production'
    ;

        // menyimpan access token ke dalam cookie
        // res.cookie('accessToken', accessToken, { ... });

```

```
        // menyimpan refresh token ke dalam cookie
        // res.cookie('refreshToken', refreshToken, { ... });


        // mengembalikan response login berhasil
    } catch (error) {
        const errorMessage = error instanceof HttpError ? error.
    message : 'Internal server error';
        const statusCode = error instanceof HttpError ? error.
    statusCode : 500;

        // mengembalikan response error internal
        return;
    }
};
```

Kode 3.5: Cuplikan Kode - Login Controller

Berdasarkan Cuplikan Kode di atas, proses login dimulai dengan validasi terhadap data yang dikirimkan oleh pengguna melalui *request body*. Validasi ini dilakukan menggunakan skema yang telah ditentukan sebelumnya untuk memastikan bahwa parameter masukan sesuai dengan format yang diharapkan. Apabila proses validasi gagal, sistem akan mengembalikan respons yang menyatakan bahwa parameter yang diberikan tidak valid.

Jika proses validasi berhasil, permintaan akan diteruskan ke komponen layanan, yaitu *login service*, yang bertugas memverifikasi kredensial pengguna. Setelah proses verifikasi selesai dan pengguna dinyatakan sah, layanan ini akan menghasilkan token otorisasi, berupa *access token* dan *refresh token*. Selain itu, skema penanganan kesalahan dirancang untuk mencatat serta mengelola kesalahan *internal server*, dengan mengembalikan pesan kesalahan yang sesuai kepada klien.

Selanjutnya, cuplikan kode 3.6 berikut memperlihatkan implementasi dari fungsi layanan *login service*, yang bertanggung jawab dalam proses autentikasi dan pembuatan token otorisasi bagi pengguna yang berhasil diverifikasi.

```
export const loginService = async (userData: UserLogin) => {
    try {
        const user = await prisma.user.findUnique({
            where: {
                username: userData.username,
            },
        });

```

```
9         if (!user) {
10            throw new HttpError('Username atau password salah',
    401);
11        }
12
13        const isPasswordValid = await bcrypt.compare(userData.
    password, user.password);
14
15        if (!isPasswordValid) {
16            throw new HttpError('Username atau password salah',
    401);
17        }
18
19        const accessToken = jwt.sign(
20            { username: user.username, role: user.role },
21            env.JWT_SECRET_ACCESS,
22            {
23                expiresIn: env.JWT_SECRET_ACCESS_LIFETIME as ms.
    StringValue,
24            },
25        );
26
27        const refreshToken = jwt.sign(
28            { username: user.username, role: user.role },
29            env.JWT_SECRET_REFRESH,
30            {
31                expiresIn: env.JWT_SECRET_REFRESH_LIFETIME as ms.
    StringValue,
32            },
33        );
34
35        return { accessToken, refreshToken };
36    } catch (error) {
37        if (error instanceof HttpError) {
38            throw error;
39        }
40        throw new HttpError('Internal Server Error', 500);
41    }
42 };
```

Kode 3.6: Cuplikan Kode - Login Service

Fungsi *login service* berfungsi sebagai unit logika bisnis yang menangani proses autentikasi berdasarkan data kredensial yang telah divalidasi. Tahapan

pertama dalam proses ini adalah pencarian data pengguna di basis data menggunakan nama pengguna yang diberikan. Jika pengguna tidak ditemukan, sistem akan langsung menolak permintaan *login* dan memberikan pesan kesalahan bahwa nama pengguna atau kata sandi tidak valid.

Langkah berikutnya adalah melakukan verifikasi terhadap kecocokan kata sandi dengan menggunakan *hashing algorithm* yang disebut bcrypt. Apabila kata sandi yang dimasukkan sesuai dengan yang tersimpan di basis data, maka sistem akan melanjutkan proses untuk menghasilkan dua buah token, yaitu *access token* dan *refresh token*. Kedua token ini dibuat dengan memanfaatkan pustaka JWT, dan masing-masing ditandatangani dengan kunci rahasia serta memiliki waktu kedaluwarsa tertentu yang ditentukan dalam konfigurasi lingkungan.

Hasil dari fungsi layanan ini berupa sepasang token otorisasi yang kemudian akan dikembalikan ke *controller* untuk digunakan dalam proses autentikasi lanjutan, seperti penyimpanan di sisi klien atau penyertaan dalam permintaan selanjutnya. Jika terjadi kesalahan selama proses, sistem akan memeriksa apakah kesalahan tersebut merupakan *instance* dari `HttpError`, dan jika tidak, maka kesalahan akan dikategorikan sebagai kesalahan internal sistem.

B Register New User

Proses registrasi pengguna baru merupakan langkah awal untuk memberikan akses resmi kepada individu dalam sistem. Dalam implementasi ini, proses registrasi dilakukan melalui dua komponen utama, yaitu *controller* dan *service*. Komponen *controller* bertanggung jawab untuk menangani permintaan yang datang dari sisi klien, melakukan validasi awal terhadap data masukan, dan meneruskan data yang telah tervalidasi ke komponen layanan (*service*) untuk diproses lebih lanjut. Cuplikan kode 3.7 berikut menunjukkan bagian dari fungsi *controller* yang menangani pendaftaran pengguna baru.

```
1  try {
2    // validasi data pendaftaran dari request body
3    const validate = await userRegisterSchema.safeParseAsync(req.
     body);
4
5    if (!validate.success) {
6      const parsedError = parseZodError(validate.error);
7
8      // mengembalikan response invalid parameter
9      return;
```

```
    }

    // memanggil service untuk menyimpan data user baru
    await registerService(validate.data);

    // mengembalikan response bahwa data berhasil dibuat
  } catch (error) {
    const errorMessage = error instanceof HttpError ? error.
    message : 'Internal server error';
    const statusCode = error instanceof HttpError ? error.
    statusCode : 500;

    // mengembalikan response error internal
    return;
  }
```

Kode 3.7: Cuplikan Kode - User Register Controller

Pada cuplikan kode *controller* di atas, proses dimulai dengan melakukan validasi terhadap data yang diterima dari pengguna melalui *request body*. Validasi dilakukan dengan menggunakan skema yang telah ditentukan sebelumnya untuk memastikan bahwa setiap parameter sesuai dengan persyaratan sistem. Apabila validasi gagal, sistem akan memberikan tanggapan yang menyatakan bahwa parameter tidak valid.

Jika seluruh data lolos validasi, maka sistem akan meneruskan informasi tersebut ke fungsi layanan *service function* yang bertugas untuk menangani logika bisnis terkait penyimpanan data pengguna baru. Setelah proses registrasi berhasil, sistem akan mengembalikan respons yang menyatakan bahwa data pengguna berhasil dibuat. Di sisi lain, apabila terjadi kesalahan pada saat eksekusi, sistem telah menerapkan mekanisme penanganan kesalahan yang akan mengembalikan pesan yang sesuai berdasarkan jenis kesalahan yang terjadi, baik kesalahan dari pengguna maupun kesalahan internal sistem.

Fungsi layanan yang bertugas memproses pendaftaran pengguna baru ditampilkan pada cuplikan kode 3.8 berikut.

```

  try {
    // memeriksa apakah user dengan username yang sama sudah ada
    const existingUser = await prisma.user.findUnique({
      where: {
        username: userData.username,
      },
```

```
    });

    if (existingUser) {
      // melempar error jika data user sudah ada
      throw new HttpError('Data already exists', 409);
    }

    // melakukan hashing terhadap password sebelum disimpan
    // const hashedPassword = bcrypt.hash(...);

    // menyimpan user baru ke dalam database
    // const newUser = prisma.user.create({ ... });

    // mengembalikan data user yang baru dibuat
    return newUser;
  } catch (error) {
    if (error instanceof HttpError) {
      throw error;
    }

    // melempar error internal jika terjadi kesalahan
    throw new HttpError('Internal Server Error', 500);
  }
```

Kode 3.8: Cuplikan Kode - User Register Service

Pada sisi layanan, fungsi *register service* memiliki tanggung jawab utama untuk memproses data pengguna yang telah divalidasi. Langkah pertama yang dilakukan adalah memeriksa apakah terdapat pengguna lain yang telah terdaftar dengan nama pengguna yang sama. Jika ditemukan data duplikat, maka sistem akan segera menghentikan proses dan mengembalikan pesan kesalahan dengan status `conflict (409)`, yang menandakan bahwa data tersebut telah ada sebelumnya dalam basis data.

Jika tidak ditemukan konflik, maka sistem akan melanjutkan ke tahap berikutnya, yaitu melakukan proses *hashing* terhadap kata sandi pengguna. Meskipun pada cuplikan kode ini bagian tersebut masih dikomentari, hal ini menunjukkan bahwa sistem telah dipersiapkan untuk menerapkan praktik keamanan yang baik dengan mengenkripsi kata sandi sebelum disimpan. Setelah proses *hashing* selesai, data pengguna yang baru akan disimpan ke dalam basis data menggunakan metode yang sesuai, dan data hasil penyimpanan akan dikembalikan sebagai hasil dari fungsi layanan.

Sebagaimana pada *controller*, fungsi layanan ini juga dilengkapi dengan mekanisme penanganan kesalahan. Jika terjadi kesalahan selama proses eksekusi, sistem akan memverifikasi apakah kesalahan tersebut merupakan bagian dari kelas `HttpError`. Jika tidak, maka kesalahan akan dikategorikan sebagai kesalahan internal dan sistem akan memberikan tanggapan sesuai dengan standar penanganan kesalahan yang telah ditentukan.

C Log Out User

Berikut adalah cuplikan *controller* untuk proses *log out* pengguna, seperti terlihat pada kode 3.9. *Controller* ini bertanggung jawab untuk menangani permintaan *log out* dari sisi server.

```
export const logoutController = async (req: Request, res: Response
    ): Promise<void> => {
  // menghapus cookie access token
  // res.clearCookie('accessToken', { ... });

  // menghapus cookie refresh token
  // res.clearCookie('refreshToken', { ... });

  // mengembalikan response logout berhasil
};
```

Kode 3.9: Cuplikan Kode - Log Out Controller

Kode ini juga menyertakan langkah-langkah untuk menghapus `cookie accessToken` dan `refreshToken`, yang berfungsi untuk menghilangkan sesi autentikasi pengguna dari *client* dengan memanggil fungsi `res.clearCookie()`. Terakhir, *controller* ini mengembalikan respons yang menandakan bahwa proses *log out* telah berhasil.

D Refresh Token

Berikut adalah *function* `refreshTokenController` seperti yang ditampilkan pada kode 3.10. Fungsi ini bertanggung jawab untuk melakukan proses *refresh token* akses pengguna agar tetap dapat mengakses sistem tanpa harus melakukan login ulang.

```
export const refreshTokenController = async (req: Request, res:
    Response): Promise<void> => {
  try {
```

```
    const refreshToken = req.cookies['refreshToken'];
    if (!refreshToken) {
      // kirim response 401
      return;
    }

    const newAccessToken = await refreshTokenService(refreshToken)
    ;

    res.cookie('accessToken', newAccessToken, {
      httpOnly: true,
      secure: true,
      maxAge: ms(env.JWT_SECRET_ACCESS_LIFETIME as ms.StringValue)
    ,
      sameSite: 'strict',
    });

    // kirim response berhasil
  } catch (error) {
    // kirim response error
  }
};
```

Kode 3.10: Cuplikan Kode - Refresh Token Controller

Fungsi `refreshTokenController` bekerja dengan memastikan terlebih dahulu bahwa `refresh token` tersedia di dalam `cookie` permintaan. Jika `refresh token` tidak ditemukan, maka sistem akan mencatat kesalahan dan mengembalikan respons dengan status `unauthorized (401)`. Jika token tersedia, fungsi akan memanggil `refreshTokenService` untuk menghasilkan `access token` baru. Token baru ini kemudian disimpan kembali ke dalam `cookie` melalui `res.cookie()` dengan beberapa pengaturan keamanan seperti `httpOnly`, `secure`, `sameSite`, dan `maxAge`. Terakhir, sistem mencatat bahwa proses berhasil dan mengirimkan respons ke *client*.

Proses pembentukan access token baru ditangani oleh fungsi `refreshTokenService`, seperti ditunjukkan pada kode 3.11. Fungsi ini memverifikasi `refresh token` menggunakan `jwt.verify` dan menghasilkan token akses berdasarkan informasi yang berhasil *didekode*.

```
export const refreshTokenService = async (refreshToken: string) =>
    {
  try {
```

```
    const decoded = jwt.verify(refreshToken, env.
    JWT_SECRET_REFRESH) as {
      username: string;
      role: string;
    };

    if (!decoded) {
      throw new HttpError('Access denied. Please log in first',
    401);
    }

    const accessToken = jwt.sign(
      { username: decoded.username, role: decoded.role },
      env.JWT_SECRET_ACCESS,
      {
        expiresIn: env.JWT_SECRET_ACCESS_LIFETIME as ms.
    StringValue,
      },
    );

    return accessToken;
  } catch (error) {
    // tangani error JWT atau HttpError
    throw new HttpError('Access denied or internal server error',
    401);
  }
};
```

Kode 3.11: Cuplikan Kode - Refresh Token Service

Fungsi `refreshTokenService` bertindak sebagai logika bisnis utama untuk memproses `refresh token`. Fungsi ini dimulai dengan memverifikasi isi dari `refresh token` menggunakan library `jwt.verify`, dengan kunci rahasia yang disediakan dalam variabel lingkungan `JWT_SECRET_REFRESH`. Setelah token berhasil diverifikasi, nilai `username` dan `role` diekstraksi dari hasil *decoding*. Jika verifikasi gagal, maka fungsi akan melempar `HttpError` dengan pesan penolakan akses.

Selanjutnya, token akses baru dibuat dengan menggunakan `jwt.sign`, yang memasukkan *username* dan *role* ke dalam *payload*. Token ini ditandatangani dengan kunci `JWT_SECRET_ACCESS` dan diatur masa berlakunya sesuai dengan `JWT_SECRET_ACCESS_LIFETIME`. Jika seluruh proses berhasil, token akses baru dikembalikan sebagai hasil dari fungsi. Namun, jika terjadi kesalahan dalam proses verifikasi atau penandatanganan, fungsi akan melempar `HttpError` yang

menyatakan akses ditolak atau terjadi kesalahan internal.

E Get Profile

Berikut merupakan *controller* `get profile` yang digunakan untuk mengembalikan data profil pengguna. Cuplikan kode ditampilkan pada 3.12.

```
export const profileController = async (req: Request, res:
    Response): Promise<void> => {
 //return response authentication success to frontend
};
```

Kode 3.12: Cuplikan Kode - Profile Controller

Fungsi `profileController` seperti yang ditunjukkan pada kode 3.12, bertugas untuk mengembalikan respons *autentikasi* yang berhasil kepada *frontend*. Logika utama pengambilan data profil tidak dilakukan di dalam *controller* ini, melainkan sudah ditangani sebelumnya oleh *middleware* JWT pada bagian 3.3. Pada tahap tersebut, data pengguna telah di-*query* dari basis data dan diteruskan menggunakan fungsi `next()`. *Controller* ini hanya menerima data yang telah tersedia dan meneruskannya sebagai respons ke *client*.

F Get All User

Berikut merupakan *controller* dan *endpoint* `Get All User` yang berfungsi untuk mengambil seluruh data pengguna. Cuplikan kode ditampilkan pada 3.13.

```
export const getAllUserController = async (req: Request, res:
    Response): Promise<void> => {
  try {
    const users = await getAllUserService();

    responseHelper(res, 'success', 200, 'Data successfully
    retrieved', users);
  } catch (error) {
    // kirim response error
  }
};
```

Kode 3.13: Cuplikan Kode - Get All User Controller

Fungsi `getAllUserController` pada 3.13 bertugas untuk memanggil `getAllUserService` guna mengambil data seluruh pengguna dari sistem. Jika proses berhasil, maka respons dikirim menggunakan fungsi `responseHelper()`

dengan status berhasil dan data pengguna yang diperoleh. Apabila terjadi kesalahan dalam proses, sistem akan mencatat kesalahan dan mengirimkan respons error yang sesuai.

Cuplikan kode `getAllUserService` yang dipanggil oleh *controller* ditampilkan pada 3.14 berikut.

```
export const getAllUserService = async () => {
  try {
    const users = await prisma.user.findMany({
      where: { deleted_at: null },
    });

    if (!users) {
      throw new HttpError('User tidak ditemukan', 404);
    }

    const parsedUsers = users.map(user => userPublicSchema.parse(
    user));

    return parsedUsers;
  } catch (error) {
    // lempar kembali HttpError atau error 500
    throw new HttpError('Internal Server Error', 500);
  }
};
```

Kode 3.14: Cuplikan Kode - Get All User Service

Fungsi `getAllUserService` yang ditunjukkan pada kode 3.14 akan menjalankan *query* ke basis data melalui `prisma.user.findMany`, dengan kondisi pengguna belum dihapus (`'deleted_at: null'`). Hasil *query* kemudian diperiksa — jika tidak ada pengguna ditemukan, maka akan dilempar `HttpError` dengan `status 404`. Jika data tersedia, maka seluruh entri akan diproses menggunakan `userPublicSchema.parse()` untuk memastikan struktur data sesuai sebelum dikembalikan sebagai hasil. Jika terjadi kesalahan dalam proses, fungsi ini akan melempar `HttpError` dengan `status 500`.

G Get User By ID

Berikut merupakan *controller Get User By ID* yang ditampilkan pada 3.15.

```
export const getUserByIdController = async (req: Request, res:
    Response): Promise<void> => {
```

```
  try {
    const userId = req.params.id;

    if (!userId) {
      // kirim response 400
      return;
    }

    const user = await getUserByIdService(userId);

    if (!user) {
      // kirim response 404
      return;
    }
    // kirim response success
  } catch (error) {
    // kirim response error
  }
};
```

Kode 3.15: Cuplikan Kode - Get User By Id Controller

Fungsi `getUserByIdController` pada 3.15 bertugas untuk mengambil data pengguna berdasarkan parameter `id` dari permintaan. Pertama, *controller* ini memeriksa apakah `userId` tersedia dalam parameter. Jika tidak tersedia, maka sistem mencatat kesalahan dan menghentikan eksekusi dengan mengirim `respons 400`. Jika `id` tersedia, maka fungsi akan memanggil `getUserByIdService`. Bila data pengguna tidak ditemukan, maka sistem akan mencatat kesalahan dan mengembalikan `status 404`. Jika proses berhasil, respons sukses akan dikirim menggunakan `responseHelper()`. Cuplikan layanan yang digunakan ditampilkan pada 3.16.

```
export const getUserByIdService = async (user_id: string) => {
  try {
    const user = await prisma.user.findFirst({
      where: {
        AND: [{ user_id }, { deleted_at: null }],
      },
    });

    if (!user) {
      throw new HttpError('User tidak ditemukan', 404);
    }

```

```
    const parsedUser = userPublicSchema.parse(user);

    return parsedUser;
  } catch (error) {
    // lempar kembali HttpError atau error 500
    throw new HttpError('Internal Server Error', 500);
  }
};
```

Kode 3.16: Cuplikan Kode - Get User By Id Service

Fungsi `getUserByIdService` pada 3.16 menangani logika bisnis untuk mengambil satu pengguna berdasarkan `user_id`. Fungsi ini menjalankan *query* menggunakan `prisma.user.findFirst` dengan kondisi bahwa `user_id` sesuai dan data belum dihapus (`deleted_at: null`). Jika data tidak ditemukan, maka akan dilemparkan `HttpError` dengan `status 404`. Apabila data ditemukan, fungsi akan memprosesnya menggunakan `userPublicSchema.parse()` untuk memastikan struktur data valid, lalu mengembalikan hasilnya. Jika terjadi kesalahan saat *query* atau parsing, maka fungsi akan melempar `HttpError` dengan `status 500`.

H Update User

Berikut cuplikan kode dari *controller* untuk proses *update user* berdasarkan `id` yang ditunjukkan pada 3.17.

```
export const editUserByIdController = async (req: Request, res:
    Response): Promise<void> => {
  try {
    const userId = req.params.id;
    if (!userId) {
      // kirim response 400
      return;
    }

    const validate = await userUpdateSchema.safeParseAsync(req.
    body);
    if (!validate.success) {
      // kirim response 400
      return;
    }

```

```
    const updatedUser = await editUserByIdService(userId, validate
    .data);
    if (!updatedUser) {
      // kirim response 404
      return;
    }

    // kirim response berhasil
  } catch (error) {
    // kirim response error
  }
};
```

Kode 3.17: Cuplikan Kode - Edit User by Id Controller

Fungsi *controller* `editUserByIdController` pada 3.17 bertugas untuk menangani permintaan perubahan data pengguna berdasarkan `id`. Pertama, *controller* akan mengambil nilai `userId` dari parameter permintaan dan melakukan validasi terhadap data masukan dengan skema *userUpdateSchema* menggunakan Zod. Jika validasi gagal, maka respons 400 dikirim. Jika data valid, fungsi akan memanggil *editUserByIdService* untuk melakukan proses pembaruan. Apabila pengguna tidak ditemukan, sistem akan mengembalikan `status 404`. Jika berhasil, sistem mencatat keberhasilan dan mengirimkan respons sukses. Cuplikan kode layanan ditampilkan pada 3.18.

```
export const editUserByIdService = async (user_id: string,
    userData: UserUpdate) => {
  try {
    const user = await prisma.user.findFirst({
      where: {
        AND: [{ user_id }, { deleted_at: null }],
      },
    });

    if (!user) {
      throw new HttpError('User tidak ditemukan', 404);
    }

    const updatedUser = await prisma.user.update({
      where: { user_id },
      data: {
        username: userData.username,
        role: userData.role,
      },
```

```
    });

    const parsedUser = userPublicSchema.parse(updatedUser);

    return parsedUser;
  } catch (error) {
    // lempar kembali HttpError atau error 500
    throw new HttpError('Internal Server Error', 500);
  }
};
```

Kode 3.18: Cuplikan Kode - Edit User By Id Service

Fungsi `editUserByIdService` pada 3.18 merupakan bagian dari logika bisnis untuk memperbarui data pengguna. Pertama, sistem akan mencari pengguna berdasarkan `id` dan memastikan data tersebut belum dihapus dengan memeriksa nilai `deleted_at`. Jika pengguna tidak ditemukan, maka fungsi akan melempar `HttpError` dengan `status 404`. Jika ditemukan, fungsi akan memperbarui data seperti `username` dan `role` menggunakan metode `prisma.user.update`. Hasil pembaruan kemudian divalidasi ulang menggunakan *userPublicSchema.parse()* sebelum dikembalikan. Jika terjadi kesalahan saat proses pembaruan atau validasi, maka akan dilempar `HttpError` dengan `status 500`.

I Create New Client

Berikut merupakan cuplikan kode untuk *controller* proses pembuatan klien baru yang ditunjukkan pada 3.19.

```
export const createClientController = async (req: Request, res:
    Response): Promise<void> => {
  const validate = await clientRequestSchema.safeParseAsync(req.
    body);

  if (!validate.success) {
    //  dan kirim response 400
    return;
  }

  try {
    const newClient = await createClientService(validate.data);


    // kirim response success
```

```
  } catch (error) {
    // kirim response error
  }
};
```

Kode 3.19: Cuplikan Kode - Create New Client Controller

Fungsi `createClientController` pada 3.19 bertugas untuk menangani permintaan pembuatan entitas klien baru. Pertama, data yang dikirim oleh pengguna akan divalidasi menggunakan skema `clientRequestSchema`. Jika validasi gagal, maka sistem akan mengembalikan `respons 400`. Jika validasi berhasil, fungsi akan memanggil *service* `createClientService` untuk melanjutkan proses bisnis. Jika proses berhasil, sistem akan mengembalikan respons sukses; sebaliknya, jika terjadi kesalahan, sistem akan mengirimkan *respons error*.

Berikut merupakan cuplikan kode *service* untuk membuat data klien baru, seperti ditunjukkan pada 3.20.

```
export const createClientService = async (clientData:
    ClientRequest) => {
  try {
    const existingClient = await prisma.client.findUnique({
      where: { client_name: clientData.client_name },
    });

    if (existingClient) {
      throw new HttpError('Client already exists', 409);
    }

    const client = await prisma.client.create({ data: clientData
    });

    return client;
  } catch (error) {
    // lempar kembali HttpError atau error 500
    throw new HttpError('Internal Server Error', 500);
  }
};
```

Kode 3.20: Cuplikan Kode - Create Client Service

Fungsi `createClientService` pada 3.20 merupakan bagian dari logika bisnis yang menangani proses pembuatan data klien baru. Fungsi ini pertama-tama akan memeriksa apakah sudah terdapat klien dengan nama yang sama melalui *query*

`prisma.client.findUnique`. Jika klien sudah ada, maka fungsi akan melempar `HttpError` dengan status `konflik (409)`. Jika tidak ditemukan, maka proses pembuatan klien akan dilanjutkan dengan `prisma.client.create`. Jika proses berhasil, data klien yang baru akan dikembalikan. Jika terjadi kesalahan selama proses berlangsung, maka sistem akan melempar `HttpError` dengan `status 500` sebagai indikasi kesalahan internal.

J Get All Client

Berikut merupakan cuplikan kode untuk *controller* pengambilan seluruh data klien seperti ditunjukkan pada 3.21.

```
export const getAllClientController = async (req: Request, res:
    Response): Promise<void> => {
  try {
    const clients = await getAllClientService();

    if (!clients || clients.length === 0) {
      //kirim response 200 tanpa data
      return;
    }

    //kirim response 200 dengan data
  } catch (error) {
    // kirim response error
  }
};
```

Kode 3.21: Cuplikan Kode - Get All Client Controller

Fungsi `getAllClientController` pada 3.21 bertugas untuk menangani permintaan pengambilan semua data klien dari sistem. Di dalamnya, fungsi ini akan memanggil *service* `getAllClientService` untuk mengambil data dari basis data. Jika tidak ditemukan data atau hasil pencarian kosong, maka akan dikembalikan respons dengan status sukses namun tanpa isi data. Jika data tersedia, maka akan dikembalikan bersama respons sukses. Jika terjadi kesalahan, akan dikirimkan `respons error`.

Cuplikan kode layanan untuk mengambil seluruh data klien dapat dilihat pada 3.22.

```
export const getAllClientService = async () => {
  try {
```

```
    const clients = await prisma.client.findMany();
    return clients;
  } catch (error) {
    // lempar kembali HttpError atau error 500
    throw new HttpError('Internal Server Error', 500);
  }
};
```

Kode 3.22: Cuplikan Kode - Get All Client Service

Fungsi `getAllClientService` pada 3.22 merupakan bagian dari logika bisnis yang digunakan untuk mengambil seluruh entri data klien dari basis data menggunakan `prisma.client.findMany()`. Jika proses pengambilan data berhasil, maka daftar klien akan dikembalikan. Namun, jika terjadi kesalahan selama proses pengambilan data, maka fungsi ini akan melemparkan `HttpError` dengan status 500 untuk menandai kesalahan internal.

K Get Client By Id

Fungsi ini digunakan untuk mengambil data *client* berdasarkan `client_id` yang dikirim melalui parameter URL. *Endpoint* ini akan memvalidasi keberadaan `id`, lalu meneruskan permintaan ke *service layer* untuk mengambil data dari *database*. Cuplikan kode *controllernya* dapat dilihat pada listing 3.23.

```
export const getClientByIdController = async (req: Request, res:
    Response): Promise<void> => {
  const clientId = req.params.id;

  if (!clientId) {
    // kirim response 400
    return;
  }

  try {
    const client = await getClientByIdService(clientId);

    if (!client) {
      // kirim response 404
      return;
    }

    // kirim response 200 dengan data client
  } catch (error) {
```

```
    // kirim response error
  }
};
```

Kode 3.23: Cuplikan Kode - Get Client By Id Controller

Kode ini berfungsi untuk mengatur alur permintaan dari *client* ke server. Validasi pertama dilakukan terhadap parameter `clientId`. Jika tidak valid, akan dikembalikan respons kesalahan. Bila valid, *controller* akan memanggil *service* untuk mengambil data *client* dari *database*.

Selanjutnya, cuplikan dari *service* yang menangani pengambilan data *client* dari *database* dapat dilihat pada listing 3.24.

```
export const getClientByIdService = async (client_id: string) => {
  try {
    const client = await prisma.client.findUnique({
      where: { client_id },
    });

    if (!client) {
      throw new HttpError('Client not found', 404);
    }

    return client;
  } catch (error) {
    // lempar kembali HttpError atau error 500
    throw new HttpError('Internal Server Error', 500);
  }
};
```

Kode 3.24: Cuplikan Kode - Get Client By Id Service

Service ini menggunakan Prisma ORM untuk mencari data *client* berdasarkan `client_id`. Jika *client* tidak ditemukan, maka akan dilemparkan `error 404`. Bila ada kesalahan sistem lainnya, akan dilempar `error 500`. Data yang ditemukan akan dikembalikan ke *controller* untuk diteruskan ke *response* API.

L Update Client By Id

Berikut merupakan cuplikan kode untuk controller `editClientByIdController` seperti ditunjukkan pada 3.25.

```
export const editClientByIdController = async (req: Request, res:
    Response): Promise<void> => {
```

```
  const clientId = req.params.id;

  if (!clientId) {
    // log error dan kirim response 400 (ID tidak valid)
    return;
  }

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

  if (!validate.success) {
    // response 400 (body tidak valid)
    return;
  }

  try {
    const updatedClient = await editClientByIdService(clientId,
    validate.data);
    //  kirim response 200 dengan data hasil update
  } catch (error) {
    // kirim response error
  }
};
```

Kode 3.25: Cuplikan Kode - Update Client By Id Controller

Fungsi `editClientByIdController` pada 3.25 bertugas memproses permintaan perubahan data klien berdasarkan `id`. Pertama-tama, fungsi memvalidasi parameter `clientId` dan melakukan validasi terhadap isi `request body` menggunakan `clientRequestSchema`. Jika validasi gagal, maka dikirimkan `response 400`. Jika berhasil, maka fungsi akan memanggil layanan `editClientByIdService` untuk memproses pembaruan data. Hasil pembaruan kemudian dikirimkan kembali ke `client` sebagai *response* sukses, atau *response error* jika terjadi kegagalan dalam proses.

Cuplikan kode layanan untuk memperbarui data *client* dapat dilihat pada 3.26 berikut.

```
export const editClientByIdService = async (client_id: string,
    clientData: ClientRequest) => {
  try {
    // cek apakah client dengan ID tersebut ada
    const client = await prisma.client.findUnique({ where: {
    client_id } });
```

```
5
6     if (!client) {
7       throw new HttpError('Client not found', 404);
8     }
9
10    // update data client
11    const updatedClient = await prisma.client.update({
12      where: { client_id },
13      data: clientData,
14    });
15
16    return updatedClient;
17  } catch (error) {
18    // lempar ulang HttpError atau error 500
19    throw new HttpError('Internal Server Error', 500);
20  }
21 };
```

Kode 3.26: Cuplikan Kode - Update Client By Id Service

Fungsi `editClientByIdService` pada 3.26 bertugas untuk menjalankan logika bisnis pembaruan data klien berdasarkan `id`. Langkah pertama yang dilakukan adalah memverifikasi apakah klien dengan `id` tersebut ada di dalam basis data. Jika tidak ditemukan, maka fungsi akan melempar `HttpError` dengan `status 404`. Jika ditemukan, maka data klien akan diperbarui dengan data yang sudah tervalidasi, lalu hasilnya dikembalikan. Bila terjadi kesalahan saat proses berlangsung, maka fungsi akan melempar `error 500` sebagai penanda kegagalan internal server.

M Create New Invoice

Berikut merupakan cuplikan kode untuk `createInvoiceController` seperti ditunjukkan pada 3.27.

```
1 export const createInvoiceController = async (req: Request, res:
    Response): Promise<void> => {
2   try {
3     // validasi body dengan Zod schema
4     const validate = await invoiceWithDetailsCreateSchema.
    safeParseAsync(req.body);
5
6     if (!validate.success) {
7       // kirim response 400 (parameter tidak valid)
```

```
      return;
    }

    // buat invoice baru lewat service
    const invoice = await createInvoiceService(validate.data);

    // response 201 dengan data invoice
  } catch (error) {
    // kirim response sesuai jenis error
  }
};
```

Kode 3.27: Cuplikan Kode - Create Invoice Controller

Fungsi `createInvoiceController` bertugas untuk menerima permintaan pembuatan *invoice* baru. Pertama-tama, fungsi ini memvalidasi data yang dikirim melalui `request body` menggunakan skema Zod `invoiceWithDetailsCreateSchema`. Apabila validasi gagal, `response 400` akan dikembalikan ke pengguna. Jika validasi berhasil, fungsi kemudian meneruskan data ke `createInvoiceService` untuk diproses lebih lanjut. Bila proses berhasil, maka *invoice* baru dikembalikan sebagai *response* sukses.

Berikut adalah cuplikan kode layanan `createInvoiceService` seperti ditunjukkan pada 3.28.

```
export const createInvoiceService = async (invoiceData:
    InvoiceWithDetailsRequest) => {
  try {
    // Validasi invoice_number tidak duplikat
    const isInvoiceNumberExists = await prisma.invoice.findUnique
    ({
      where: { invoice_number: invoiceData.invoice_number },
    });

    if (isInvoiceNumberExists) {
      throw new HttpError('Duplicate invoice', 409);
    }

    // Validasi client_id harus ada
    const isClientExists = await prisma.client.findUnique({
      where: { client_id: invoiceData.client_id },
    });

    if (!isClientExists) {
      throw new HttpError('Client not found', 404);
```

```
    }

    // Hitung amount setiap detail = delivery_count * price_per_delivery
    const invoiceDetailsWithAmount = invoiceData.invoice_details.map(detail => ({
      ...detail,
      amount: detail.delivery_count * detail.price_per_delivery,
    }));

    // Hitung:
    // - subTotal
    // - taxRate
    // - taxAmount
    // - total

    const subTotal = ...;
    const taxRate = ...;
    const taxAmount = ...;
    const total = ...;

    const amountPaid = 0;
    const paymentStatus = 'unpaid';

    // Simpan invoice dan detail-nya secara bersamaan menggunakan transaction
    const createdInvoice = await prisma.$transaction(async tx => {
      const invoice = await tx.invoice.create({
        data: {...},
      });

      await tx.invoiceDetail.createMany({
        data: invoiceDetailsWithAmount.map(detail => ({
          ...detail,
          invoice_id: invoice.invoice_id,
        })),
      });

      return invoice;
    });

    return createdInvoice;
  } catch (error) {
```

```
    if (error instanceof HttpError) throw error;
    throw new HttpError('Internal Server Error', 500);
  }
};
```

Kode 3.28: Cuplikan Kode - Create Invoice Service

Fungsi `createInvoiceService` bertanggung jawab atas seluruh proses logika bisnis dalam pembuatan *invoice* beserta detailnya. Pertama, dilakukan validasi apakah `invoice_number` yang dimasukkan sudah pernah digunakan. Jika iya, maka akan dilempar `HttpError` dengan status `409 (conflict)`. Selanjutnya, sistem memastikan bahwa `client_id` yang dikaitkan memang ada di dalam *database*. Setelah validasi selesai, sistem menghitung nilai `amount` untuk masing-masing *item detail* berdasarkan jumlah pengiriman (`delivery_count`) dan harga per pengiriman (`price_per_delivery`).

Langkah berikutnya adalah menghitung *subtotal*, nilai pajak, dan total keseluruhan *invoice*. Seluruh data *invoice* beserta detailnya kemudian disimpan secara atomik menggunakan transaksi `prisma.$transaction`, untuk memastikan konsistensi data antara *invoice* utama dan detailnya. Jika semua proses berjalan sukses, maka *invoice* yang telah dibuat dikembalikan. Apabila terjadi kesalahan, *error* yang relevan akan dilempar untuk ditangani oleh *controller*.

N Get All Invoice

Berikut merupakan cuplikan kode untuk controller `getAllInvoiceController` seperti ditunjukkan pada 3.29.

```
export const getAllInvoiceController = async (req: Request, res:
    Response): Promise<void> => {
  try {
  const invoices = await getAllInvoiceService();

  const message = (!invoices || invoices.length === 0)
    ? 'No content to display'
    : 'Data successfully retrieved';

    // return response 200 ke frontend
    } catch (error) {

      const errorMessage = error instanceof HttpError ? error.
    message : 'Internal server error';
```

```
      const statusCode = error instanceof HttpError ? error.
    statusCode : 500;

      // return response error ke frontend
    }
};
```

Kode 3.29: Cuplikan Kode - Get All Invoice Controller

Fungsi `getAllInvoiceController` bertugas untuk menangani permintaan pengambilan seluruh data *invoice*. Fungsi ini memanggil `getAllInvoiceService` untuk memperoleh data *invoice* beserta detail dan data *client* terkait. Apabila data kosong, *controller* tetap mengembalikan *response* dengan pesan `"No content to display"`. Jika data ditemukan, maka akan dikembalikan dengan pesan sukses. Bila terjadi kesalahan selama proses, *error* akan ditangani dan *response* akan dikirim sesuai jenis *error*-nya.

Berikut adalah cuplikan kode layanan `getAllInvoiceService` seperti ditunjukkan pada 3.30.

```
export const getAllInvoiceService = async () => {
  try {
    const invoices = await prisma.invoice.findMany({
      include: {
        client: {
          select: {
            client_id: true,
            client_name: true,

          },
        },
        invoice_details: {
          select: {
            delivery_count: true,
            price_per_delivery: true,
            amount: true,
          },
        },
      },
      orderBy: {
        issue_date: 'desc',
      },
    });


```

```
    return invoices;
  } catch (error) {
    if (error instanceof HttpError) {
      throw error;
    }

    throw new HttpError('Internal Server Error', 500);
  }
```

Kode 3.30: Cuplikan Kode - Get All Invoice Service

Fungsi `getAllInvoiceService` menjalankan proses pengambilan data *invoice* dari *database* menggunakan Prisma ORM. Data yang diambil mencakup informasi *client* yang berelasi dan daftar `invoice_details` dari masing-masing *invoice*. Data diurutkan berdasarkan tanggal terbit *invoice* (`issue_date`) secara menurun (terbaru terlebih dahulu). Jika terjadi kesalahan selama proses pengambilan data, maka *error* akan dilemparkan sebagai `HttpError` agar dapat ditangani oleh *controller*.

O Get Invoice By Id

Berikut merupakan cuplikan kode untuk *controller* `getInvoiceByIdController` seperti ditunjukkan pada 3.31.

```
export const getInvoiceByIdController = async (req: Request, res:
    Response): Promise<void> => {
  const invoiceId = req.params.id;

  if (!invoiceId) {
    // return response 400 ke frontend
    return;
  }

  try {
    const invoice = await getInvoiceByIdService(invoiceId);

    if (!invoice) {
      // return response 404 ke frontend
      return;
    }

    // return response 200 ke frontend
  } catch (error) {
```

```
    const errorMessage = error instanceof HttpError ? error.
    message : 'Internal server error';
    const statusCode = error instanceof HttpError ? error.
    statusCode : 500;

    // return response error ke frontend
  }
};
```

Kode 3.31: Cuplikan Kode - Get Invoice By Id Controller

Fungsi `getInvoiceByIdController` bertugas untuk menangani permintaan pengambilan *invoice* berdasarkan `id`. Fungsi ini akan memverifikasi parameter `id` dari `req.params`. Jika `id` tidak tersedia, maka akan dikembalikan `response 400`. Setelah itu, *controller* akan memanggil fungsi layanan `getInvoiceByIdService` untuk mengambil data *invoice* dari *database*. Jika data tidak ditemukan, maka `response 404` akan dikembalikan ke *frontend*. Bila terjadi kesalahan lainnya, *controller* akan menangani dan mengirim `response error` dengan kode dan pesan yang sesuai.

Berikut adalah cuplikan kode layanan `getInvoiceByIdService` seperti ditunjukkan pada 3.32.

```
export const getInvoiceByIdService = async (invoice_id: string) =>
    {
  try {
    // Cari invoice berdasarkan ID, termasuk relasi client dan
    invoice_details
    const invoice = await prisma.invoice.findUnique({
      where: {
        invoice_id,
      },
      include: {
        client: true,
        invoice_details: true,
      },
    });

    // Jika invoice tidak ditemukan, lempar error 404
    // throw new HttpError('Invoice not found', 404);

    // return invoice
  } catch (error) {
    // Jika error dari jenis HttpError, lempar ulang
```

```
    // Jika bukan, lempar error 500

    // throw new HttpError('Internal Server Error', 500);
  }
};
```

Kode 3.32: Cuplikan Kode - Get Invoice By Id Service

Fungsi `getInvoiceByIdService` bertugas untuk mengambil satu data *invoice* berdasarkan `invoice_id`. Proses ini juga memuat data relasi seperti informasi `client` dan daftar `invoice_details`. Jika *invoice* dengan `id` yang dimaksud tidak ditemukan, maka akan dilemparkan *error* dengan kode `status 404`. Bila terjadi *error* lain, *error* akan dilempar ulang sebagai `HttpError` dengan *status 500*. Data *invoice* yang berhasil ditemukan akan dikembalikan ke *controller* untuk diproses lebih lanjut.

P Update Invoice By Id

Berikut merupakan cuplikan kode *controller* untuk proses *update invoice* berdasarkan `id` seperti ditunjukkan pada 3.33.

```
export const updateInvoiceByIdController = async (req: Request,
    res: Response): Promise<void> => {
  try {
    const invoiceId = req.params.id;

    if (!invoiceId) {
      // return response 400 ke frontend
      return;
    }

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

    if (!validate.success) {
      const parsed = parseZodError(validate.error);

      // return response 400 ke frontend
      return;
    }

    const invoice = await updateInvoiceByIdService(invoiceId,
    validate.data);
```

```
20
21      // return response 200 ke frontend
22    } catch (error) {
23      const errorMessage = error instanceof HttpError ? error.
        message : 'Internal server error';
24      const statusCode = error instanceof HttpError ? error.
        statusCode : 500;
25
26      // return response error ke frontend
27    }
28  };
```

Kode 3.33: Cuplikan Kode - Update Invoice By Id Controller

Fungsi `updateInvoiceByIdController` bertugas untuk menangani permintaan update data *invoice* berdasarkan `id`. Pertama-tama, *controller* akan memverifikasi keberadaan *id invoice* pada `req.params`. Setelah itu, isi `body` dari *request* akan divalidasi menggunakan skema Zod. Bila data valid, maka fungsi akan melanjutkan pemanggilan *service* `updateInvoiceByIdService`. Jika terjadi *error* saat validasi atau proses lainnya, `response error` akan dikembalikan ke *frontend* sesuai jenis *error*-nya.

Selanjutnya, berikut adalah cuplikan kode untuk *service update invoice* seperti ditunjukkan pada 3.34.

```
1  export const updateInvoiceByIdService = async (
2    invoice_id: string,
3    invoiceData: InvoiceWithDetailsUpdate,
4  ) => {
5    try {
6      const isInvoiceExist = await prisma.invoice.findUnique({
7        where: { invoice_id },
8      });
9
10     if (!isInvoiceExist) {
11       throw new HttpError('Invoice not found', 404);
12     }
13
14     const isClientExists = await prisma.client.findUnique({
15       where: { client_id: invoiceData.client_id },
16     });
17
18     if (!isClientExists) {
19       throw new HttpError('Client not found', 404);
20     }
```

```
21
22    const updatedInvoice = await prisma.$transaction(async tx => {
23      const existingDetails = await tx.invoiceDetail.findMany({
    where: { invoice_id } });
24
25      for (const detail of invoiceData.invoice_details) {
26        // hitung amount = delivery_count * price_per_delivery
27
28        if (detail.invoice_detail_id) {
29          await tx.invoiceDetail.update({ ... });
30        } else {
31          await tx.invoiceDetail.create({ ... });
32        }
33      }
34
35      // cari data detail yang dihapus lalu hapus dengan delete
36      for (const detail of existingDetails) {
37        // jika detail tidak ada di invoiceData, maka delete
38        await tx.invoiceDetail.delete({ ... });
39      }
40
41      // ambil kembali semua invoiceDetail yang baru
42      // menghitung subtotal
43      // menghitung taxRate
44      // menghitung taxAmount
45      // menghitung total
46
47      const updatedInvoice = await tx.invoice.update({ ... });
48
49      await _updateInvoiceColPaymentStatusService(tx, invoice_id);
50
51      // jika invoice_number berubah, update juga di tabel payment
52      if (isInvoiceExist.invoice_number !== invoiceData.
    invoice_number) {
53        await updatePaymentColInvoiceNumberService(tx, invoice_id,
     invoiceData.invoice_number);
54      }
55
56      return updatedInvoice;
57    });
58
59    return updatedInvoice;
60  } catch (error) {
```

```
61      if (error instanceof HttpError) throw error;
62      throw new HttpError('Internal Server Error', 500);
63    }
64  };
```

Kode 3.34: Cuplikan Kode - Update Invoice By Id Service

Fungsi `updateInvoiceByIdService` merupakan *business logic* untuk memperbarui data *invoice*. Proses ini dilakukan secara menyeluruh dengan menggunakan `transaction` agar perubahan *invoice* dan *invoice detail* dapat dijalankan secara atomik. Langkah-langkah utamanya meliputi:

- Validasi keberadaan invoice dan client.
- Memproses setiap detail invoice:
  - Update jika `invoice_detail_id` tersedia.
  - Create jika `invoice_detail_id` tidak tersedia.
- Menghapus detail yang sudah tidak ada pada data baru.
- Menghitung ulang total, pajak, dan subtotal invoice.
- Mengupdate payment status invoice.
- Mengupdate kolom `invoice_number` pada tabel `payment` jika terjadi perubahan.

Dengan penggunaan transaksi, integritas data tetap terjaga meskipun ada banyak entitas yang terlibat dalam pembaruan.

3.5.4 Zod Schema

A User Schema

Cuplikan kode berikut menunjukkan definisi skema validasi entitas *User* menggunakan pustaka *Zod*:

```
export const userSchema = z.object({
  user_id: z.string().uuid(),
  username: z.string().min(1).max(50),
  role: z.enum(['management', 'finance']),
  // Atribut lainnya disembunyikan untuk ringkasan dan
   perlindungan informasi
```

```
6
7   created_at: z.date(),
8   updated_at: z.date(),
9 });
```

Kode 3.35: Cuplikan Skema - User Schema

Skema ini memastikan bahwa data pengguna yang diterima sesuai dengan format yang telah ditentukan sebelum diproses lebih lanjut.

B Client Schema

Cuplikan kode berikut merupakan versi ringkas dari skema validasi entitas *Client* yang disusun menggunakan pustaka *Zod*:

```
1  export const clientSchema = z.object({
2    client_id: z.string().uuid(),
3    client_name: z.string().min(2).max(100),
4    currency: z.string().length(3),
5    country: z.string(),
6    // Atribut lainnya disembunyikan untuk ringkasan dan
       perlindungan informasi
7
8    createdAt: z.date(),
9    updatedAt: z.date(),
10 });
```

Kode 3.36: Cuplikan Skema - Client Schema

Skema ini digunakan untuk menjamin validitas struktur data klien sebelum disimpan ke dalam basis data.

C Invoice Schema

```
1  export const invoiceSchema = z.object({
2    invoice_id: z.string().uuid(),
3    invoice_number: z.string(),
4    issue_date: z.string(),
5    // Atribut lainnya disembunyikan untuk ringkasan dan
       perlindungan informasi
6
7    created_at: z.date(),
8    updated_at: z.date(),
9  });
```

Kode 3.37: Cuplikan Model - Invoice Detail

D Invoice Detail Schema

Cuplikan kode berikut mendefinisikan skema validasi entitas *Invoice Detail* dengan menggunakan pustaka *Zod*:

```
export const invoiceDetailSchema = z.object({
  invoice_detail_id: z.string().uuid(),
  transaction_note: z.string(),
  // Atribut lainnya disembunyikan untuk ringkasan dan
    perlindungan informasi

  createdAt: z.date(),
  updatedAt: z.date(),
});
```

Kode 3.38: Cuplikan Skema - Invoice Detail Schema

Skema ini digunakan untuk memvalidasi struktur data pada detail faktur sebelum diproses oleh sistem.

3.5.5 Prisma Model

A User Model

Berikut merupakan cuplikan kode 3.39 model *User* yang diimplementasikan menggunakan *Prisma Schema Language*.

```
model User {
  user_id    String @id @default(uuid())
  username   String @unique

  // Atribut lainnya disembunyikan untuk ringkasan dan
    perlindungan informasi
  // ...
}
```

Kode 3.39: Cuplikan Model - User Model

Model ini digunakan untuk proses sinkronisasi struktur tabel pengguna pada basis data melalui mekanisme *database migration* dengan Prisma. Struktur tersebut mencakup atribut identitas unik `user_id` dan `username` yang bersifat unik untuk setiap entri.

B Client Model

Cuplikan kode 3.40 berikut menunjukkan definisi model *Client* yang diimplementasikan menggunakan Prisma:

```
model Client {
  client_id       String    @id @default(uuid())
  client_name     String    @unique
  country         String
  // Atribut lain disembunyikan untuk ringkasan dan perlindungan
    informasi
}
```

Kode 3.40: Cuplikan Model - Client menggunakan Prisma

Model ini merepresentasikan entitas klien dalam sistem dan disinkronkan ke basis data melalui *database migration*. Atribut `client_id` berfungsi sebagai identitas unik, sementara `client_name` memiliki batasan nilai unik untuk mencegah duplikasi data.

C Invoice Model

Cuplikan kode 3.41 berikut menunjukkan definisi model *Invoice* yang diimplementasikan menggunakan Prisma:

```
model Invoice {
  invoice_id      String    @id @default(uuid())
  invoice_number  String    @unique
  issue_date      DateTime
  due_date        DateTime

  // Atribut lain disembunyikan untuk ringkasan dan perlindungan
    informasi
  // ...
}
```

Kode 3.41: Cuplikan Model - Invoice menggunakan Prisma

Model ini digunakan untuk merepresentasikan entitas faktur dalam sistem. Atribut `invoice_id` bertindak sebagai identitas unik, sementara `invoice_number` memiliki nilai unik untuk membedakan setiap entri faktur. Atribut `issue_date` dan `due_date` masing-masing merepresentasikan tanggal penerbitan dan tanggal jatuh tempo faktur.

D Invoice Detail Model

Cuplikan kode 3.42 berikut menunjukkan definisi model *InvoiceDetail* yang diimplementasikan menggunakan Prisma:

```
model InvoiceDetail {
  invoice_detail_id      String    @id @default(uuid())
  delivery_count         Int
  amount                 Float
  // Atribut lain disembunyikan untuk ringkasan dan perlindungan informasi
}
```

Kode 3.42: Cuplikan Model Invoice Detail Model

Model ini digunakan untuk merepresentasikan entitas detail faktur dalam sistem. Atribut `invoice_detail_id` berfungsi sebagai identitas unik, sedangkan `delivery_count` dan `amount` merepresentasikan data kuantitatif terkait pengiriman dan nominal transaksi.

3.6 API Contract

Dalam section ini akan disajikan beberapa endpoint API yang digunakan dalam sistem, lengkap dengan detail kontrak seperti metode HTTP, *path endpoint*, *header*, dan deskripsi singkat dari masing-masing fungsinya.

3.6.1 POST /users/login

- **Tags:** `Users`
- **Summary:** Login user
- **URL Params**
  - `None`
- **Data Params**
  - `username [string] (not null)`
  - `password [string] (not null)`
- **Headers**

- `Content-Type: application/json`

- **Success Response**
  - **Code:** `200 OK`
  - **Deskripsi:** Login berhasil, token diberikan dalam *cookie*
  - **Content:**

```
{
  "status": "success",
  "message": "Successfully logged in",
  "data": null
}
```

Kode 3.43: Response 200 - Login Berhasil

  - **Header Tambahan:**
    * `Set-Cookie`: Token akses dan *refresh* sebagai *cookie*

- **Error Response**
  - **Code:** `400 Bad Request`
    Deskripsi: Parameter tidak valid apabila `username` kosong

```
{
  "status": "error",
  "message": "Invalid parameters",
  "data": [
    {
        "error": {
            "username": [
                "Username tidak boleh kosong"
            ]
        }
    }
  ]
}
```

Kode 3.44: Response 400 - Username Kosong

  - **Code:** `401 Unauthorized`
    Deskripsi: `Username` atau `password` salah

```
{
  "status": "error",
```

```
  "message": "Username atau password salah",
  "data": null
}
```

Kode 3.45: Response 401 - Username atau Password Salah

- **Code:** `500 Internal Server Error`
  Deskripsi: Terjadi kesalahan pada server

```
{
  "status": "error",
  "message": "Internal server error",
  "data": null
}
```

Kode 3.46: Response 500 - Kesalahan Server

3.6.2 POST /users/register

- **Tags:** `Users`
- **Summary:** Mendaftarkan user baru ke dalam sistem.
- **URL Params**
  - `None`
- **Data Params**
  - `username [string] (not null)`
  - `password [string] (min_length: 8, max_length: 50)`
  - `role [string] (enum: management)`
- **Headers**
  - `Content-Type: application/json`
- **Success Response**
  - **Code:** `201 Created`
  - **Content:**

```
{
  "status": "success",
  "message": "Data successfully created",
  "data": null
}
```

Kode 3.47: Response 201 - Register Berhasil

- **Error Response**
  - **Code:** `400 Bad Request`
    Deskripsi: Parameter tidak valid apabila `username` kosong

```
{
  "status": "error",
  "message": "Invalid parameters",
  "data": [
    {
        "error": {
            "username": [
                "Username tidak boleh kosong"
            ]
        }
    }
  ]
}
```

Kode 3.48: Response 400 - Username Kosong

  - **Code:** `409 Conflict`
    Deskripsi: `User` dengan `username` yang sama sudah ada

```
{
  "status": "error",
  "message": "Data already exists",
  "data": null
}
```

Kode 3.49: Response 409 - Username Sudah Ada

  - **Code:** `500 Internal Server Error`
    Deskripsi: Terjadi kesalahan pada server

```
{
  "status": "error",
  "message": "Internal server error",
  "data": null
```

```
}
```

Kode 3.50: Response 500 - Kesalahan Server

3.6.3 DELETE /users/logout

- **Tags:** `Users`
- **Summary:** Logout user
- **URL Params**
  - `None`
- **Data Params**
  - `None`
- **Headers**
  - `Content-Type: application/json`
- **Success Response**
  - **Code:** `204 No Content`
  - **Deskripsi:** Berhasil *logout*, `cookie` dihapus
  - **Content:**

```
{
  "status": "success",
  "message": "Successfully logged out",
  "data": null
}
```

Kode 3.51: Response 204 - Logout Berhasil

  - **Header Tambahan:**
    * `Set-Cookie`: Menghapus `accessToken` dan `refreshToken`

UNIVERSITAS MULTIMEDIA NUSANTARA

3.6.4 POST /users/refresh-token

- **Tags:** `Users`
- **Summary:** Refresh access token menggunakan `refresh token` yang tersimpan di `cookie`
- **URL Params**
  - `None`
- **Data Params**
  - `None`
- **Headers**
  - `Content-Type: application/json`
  - `Cookie`: Mengandung `refresh token`
- **Success Response**
  - **Code:** `200 OK`
  - **Deskripsi:** Access token berhasil diperbarui
  - **Content:**

```
{
  "status": "success",
  "message": "Access token successfully refreshed",
  "data": null
}
```

Kode 3.52: Response 200 - Token Berhasil Diperbarui

  - **Header Tambahan:**
    - `Set-Cookie`: Access token baru dikirim sebagai cookie
- **Error Response**
  - **Code:** `401 Unauthorized`

    Deskripsi: Tidak ada refresh token atau token tidak valid

UNIVERSITAS MULTIMEDIA NUSANTARA

```
{
  "status": "error",
  "message": "Access denied. Please log in first",
  "data": null
}
```

Kode 3.53: Response 401 - Refresh Token Tidak Valid

- **Code:** `500 Internal Server Error`
  Deskripsi: Terjadi kesalahan pada server

```
{
  "status": "error",
  "message": "Internal server error",
  "data": null
}
```

Kode 3.54: Response 500 - Kesalahan Server

3.6.5 GET /users/profile

- **Tags:** `Users`
- **Summary:** Ambil data profil *user* yang sedang *login*
- **URL Params**
  - `None`
- **Data Params**
  - `None`
- **Headers**
  - `Content-Type: application/json`
  - `Cookie`: Mengandung access token yang valid
- **Success Response**
  - **Code:** `200 OK`
  - **Deskripsi:** Autentikasi berhasil dan profil berhasil diambil
  - **Content:**

```
{
  "status": "success",
  "code": 200,
  "message": "Authentication successful",
  "data": {
    "username": "miti",
    "role": "management"
  }
}
```

Kode 3.55: Response 200 - Profil User

- **Error Response**
  - **Code:** `401 Unauthorized`
    Deskripsi: User tidak login atau token tidak valid

```
{
  "status": "error",
  "message": "Access denied. Please log in first",
  "data": null
}
```

Kode 3.56: Response 401 - Tidak Terautentikasi

3.6.6 GET /users

- **Tags:** `Users`
- **Summary:** Ambil semua data *user* (khusus `management`)
- **URL Params**
  - `None`
- **Data Params**
  - `None`
- **Headers**
  - `Content-Type: application/json`
  - `Cookie`: Mengandung access token yang valid
- **Success Response**

- **Code:** `200 OK`
- **Deskripsi:** Berhasil mengambil semua data *user*
- **Content:**

```
{
  "status": "success",
  "code": 200,
  "message": "Data successfully retrieved",
  "data": [
    {
      "user_id": "0a9e4a22-8f1c-4c2a-9a3a-2c3a5d1c1234",
      "username": "miti",
      "role": "management",
      "created_at": "2024-04-29T12:34:56.000Z",
      "updated_at": "2024-04-30T08:20:00.000Z"
    }
  ]
}
```

Kode 3.57: Response 200 - Ambil Semua Data User

- **Error Response**
  - **Code:** `401 Unauthorized`
    Deskripsi: User belum login atau token tidak valid

```
    {
      "status": "error",
      "message": "Access denied. Please log in first",
      "data": null
    }

```

Kode 3.58: Response 401 - Unauthorized

  - **Code:** `403 Forbidden`
    Deskripsi: Tidak memiliki akses — hanya user dengan role `management` yang diizinkan

```
    {
      "status": "error",
      "message": "Access denied. Only management role
    allowed",
      "data": null
    }

```

Kode 3.59: Response 403 - Forbidden

- **Code:** `500 Internal Server Error`
  Deskripsi: Terjadi kesalahan pada server

```
{
  "status": "error",
  "message": "Internal server error",
  "data": null
}
```

Kode 3.60: Response 500 - Server Error

3.6.7 GET /users/{id}

- **Tags:** `Users`
- **Summary:** Ambil detail *user* berdasarkan `id`
- **URL Params**
  - `id [UUID] (required)` — ID dari user
- **Data Params**
  - `None`
- **Headers**
  - `Content-Type: application/json`
  - `Cookie`: Mengandung access token yang valid
- **Success Response**
  - **Code:** `200 OK`
  - **Content:**

```
{
  "status": "success",
  "code": 200,
  "message": "Data successfully retrieved",
  "data": {
    "user_id": "0a9e4a22-8f1c-4c2a-9a3a-2c3a5d1c1234",
    "username": "miti",
    "role": "management",
```

```
    "created_at": "2024-04-29T12:34:56.000Z",
    "updated_at": "2024-04-30T08:20:00.000Z"
  }
}
```

Kode 3.61: Response 200 - Detail User Berhasil Diambil

- **Error Response**
  - **Code:** `400 Bad Request`
    Deskripsi: Parameter `id` tidak valid

```
{
  "status": "error",
  "message": "Invalid user ID parameter",
  "data": null
}
```

Kode 3.62: Response 400 - Parameter Tidak Valid

  - **Code:** `401 Unauthorized`
    Deskripsi: User belum login atau token tidak valid

```
{
  "status": "error",
  "message": "Access denied. Please log in first",
  "data": null
}
```

Kode 3.63: Response 401 - Belum Login

  - **Code:** `403 Forbidden`
    Deskripsi: Hanya user dengan role `management` yang boleh mengakses

```
{
  "status": "error",
  "message": "Access denied. You are not authorized to
    access this resource",
  "data": null
}
```

Kode 3.64: Response 403 - Akses Ditolak

  - **Code:** `404 Not Found`
    Deskripsi: Data user tidak ditemukan

```
{
  "status": "error",
```

```
  "message": "User not found",
  "data": null
}
```

Kode 3.65: Response 404 - User Tidak Ditemukan

– **Code:** `500 Internal Server Error`
Deskripsi: Terjadi kesalahan pada server

```
{
  "status": "error",
  "message": "Internal server error",
  "data": null
}
```

Kode 3.66: Response 500 - Kesalahan Server

3.6.8 PUT /users/{id}

- **Tags:** `Users`
- **Summary:** Mengupdate data *user* berdasarkan `id`.
- **URL Params**
  - `id` (string): UUID dari user, tidak boleh `null`
- **Data Params**
  - `username` (`string`): tidak boleh kosong
  - `role` (`string`): enum, hanya dapat bernilai `management`
- **Headers**
  - `Content-Type`: `application/json`
  - `Cookie`: Mengandung access token yang valid
- **Success Response**
  - **Code:** `200 OK`
  - **Content:**

```
{
  "status": "success",
  "code": 200,
  "message": "Data successfully updated",
  "data": {
    "user_id": "291d5bf1-e0d1-41f9-8229-e7de89a6f3dd",
    "username": "tio",
    "role": "management",
    "created_at": "2025-04-30T04:57:40.706Z",
    "updated_at": "2025-04-30T10:12:35.289Z"
  }
}
```

Kode 3.67: Response 200 - Update Berhasil

- **Error Response**
  - **Code:** `400 Bad Request`
    Deskripsi: Parameter `id` tidak valid atau data `request` tidak sesuai

```
{
  "status": "error",
  "message": "Invalid request body or ID parameter",
  "data": null
}
```

Kode 3.68: Response 400 - Bad Request

  - **Code:** `401 Unauthorized`
    Deskripsi: User belum login atau token tidak valid

```
{
  "status": "error",
  "message": "Access denied. Please log in first",
  "data": null
}
```

Kode 3.69: Response 401 - Unauthorized

  - **Code:** `403 Forbidden`
    Deskripsi: Tidak memiliki akses — hanya role `management` yang diperbolehkan

```
{
  "status": "error",
```

```
  "message": "Access denied. You are not authorized to
   access this resource",
  "data": null
}
```

Kode 3.70: Response 403 - Forbidden

- **Code:** `404 Not Found`
  Deskripsi: *User* dengan ID yang dimaksud tidak ditemukan

```
{
  "status": "error",
  "message": "User not found",
  "data": null
}
```

Kode 3.71: Response 404 - Not Found

- **Code:** `500 Internal Server Error`
  Deskripsi: Terjadi kesalahan pada server

```
{
  "status": "error",
  "message": "Internal server error",
  "data": null
}
```

Kode 3.72: Response 500 - Server Error

3.6.9 POST /clients

- **Tags:** `Clients`
- **Summary:** Membuat *client* baru
- **URL Params**
  - `None`
- **Data Params**
  - `client_name [string] (not null)`
  - `currency [string] (enum: IDR, USD)`
  - `country [string] (enum: Indonesia, China, United State)`

- `client_address [string] (not null)`
- `postal_code [string] (not null)`
- `client_phone [string] (not null)`

- **Headers**
  - `Content-Type: application/json`
  - `Cookie`: Mengandung `access token` dengan *role* `finance`
- **Success Response**
  - **Code:** `201 Created`
  - **Content:**

```
{
  "status": "success",
  "message": "Data successfully created",
  "data": {
    // data client yang berhasil dibuat
  }
}
```

Kode 3.73: Response 201 - Client Berhasil Dibuat

Response: 400 Bad Request

Deskripsi: Parameter tidak valid apabila `client_name` kosong

```
{
  "status": "error",
  "message": "Invalid parameters",
  "data": [
    {
      "field": "client_name",
      "message": "Client name is required"
    }
  ]
}
```

Kode 3.74: Response 400 - Client Name Kosong

Response: 409 Conflict

Deskripsi: Client sudah ada

```
{
  "status": "error",
  "message": "Client already exists",
  "data": null
}
```

Kode 3.75: Response 409 - Client Sudah Ada

Response: 500 Internal Server Error

Deskripsi: Terjadi kesalahan pada server

```
{
  "status": "error",
  "message": "Internal server error",
  "data": null
}
```

Kode 3.76: Response 500 - Kesalahan Server

3.6.10 GET /clients

- **Tags:** `Clients`
- **Summary:** Mendapatkan semua data *client*
- **Method:** GET
- **Endpoint:** `/clients`
- **Headers:**
  - `Content-Type: application/json`
  - `Cookie`: Mengandung access token yang valid

Response: 200 OK

Deskripsi: Data client berhasil diambil.

```
{
  "status": "success",
  "code": 200,
  "message": "Data successfully retrieved",
  "data": [
    {
      "client_id": "123e4567-e89b-12d3-a456-426614174000",
      "client_name": "PT Maju Jaya",
      "currency": "USD",
      "country": "Indonesia",
      "client_address": "Jl. Sudirman No.1",
      "postal_code": "12345",
      "client_phone": "+6281234567890",
      "createdAt": "2024-01-01T00:00:00.000Z",
      "updatedAt": "2024-01-02T00:00:00.000Z"
    }
  ]
}
```


Response: 500 Internal Server Error

Deskripsi: Terjadi kesalahan pada server.


```
{
  "status": "error",
  "code": 500,
  "message": "Internal server error",
  "data": null
}
```


3.6.11 GET /clients/{id}

- **Tags:** `Clients`
- **Summary:** Mendapatkan data *client* berdasarkan `id`
- **URL Params**
  - `id [UUID] (required)` — ID unik client
- **Data Params**
  - `None`

- **Headers**
  - `Content-Type: application/json`
  - `cookies: accessToken`
- **Success Response**
  - **Code:** `200 OK`
  - **Content:**

```
{
  "status": "success",
  "code": 200,
  "message": "Data successfully retrieved",
  "data": {
    "client_id": "123e4567-e89b-12d3-a456-426614174000",
    "client_name": "PT Maju Jaya",
    "currency": "USD",
    "country": "Indonesia",
    "client_address": "Jl. Sudirman No.1",
    "postal_code": "12345",
    "client_phone": "+6281234567890",
    "createdAt": "2024-01-01T00:00:00.000Z",
    "updatedAt": "2024-01-02T00:00:00.000Z"
  }
}
```

Kode 3.77: Response 200 - Data Client Ditemukan

- **Error Response**
  - **Code:** `400 Bad Request`
    Deskripsi: Parameter `id` tidak valid

```
{
  "status": "error",
  "code": 400,
  "message": "Invalid parameters",
  "data": null
}
```

Kode 3.78: Response 400 - Parameter ID Tidak Valid

  - **Code:** `404 Not Found`
    Deskripsi: Data client tidak ditemukan berdasarkan ID

```
{
  "status": "success",
  "code": 404,
  "message": "No client found for ID: 123e4567-e89b-12d3-
    a456-426614174000",
  "data": null
}
```

Kode 3.79: Response 404 - Client Tidak Ditemukan

- **Code:** `500 Internal Server Error`
  Deskripsi: Terjadi kesalahan pada server

```
{
  "status": "error",
  "code": 500,
  "message": "Internal server error",
  "data": null
}
```

Kode 3.80: Response 500 - Kesalahan Server

3.6.12 PUT /clients/{id}

- **Tags:** `Clients`
- **Summary:** Memperbarui data klien berdasarkan `id`
- **URL Params**
  - `id [UUID] (required)` — ID dari client
- **Headers**
  - `Content-Type: application/json`
  - `Cookie`: Mengandung `access token` yang valid
- **Request Body**

```
{
  "client_name": "client 2",
  "currency": "USD",
  "country": "China",
  "client_address": "pondok indah",
  "postal_code": "10100",
```

```
  "client_phone": "62123412341234"
}
```

Kode 3.81: Request Body - Update Client

- **Success Response**
  - **Code:** `200 OK`
  - **Deskripsi:** Data client berhasil diperbarui

```
{
  "status": "success",
  "code": 200,
  "message": "Data successfully updated",
  "data": {
    "client_id": "123e4567-e89b-12d3-a456-426614174000",
    "client_name": "apkwab",
    "currency": "USD",
    "country": "China",
    "client_address": "pondok indah",
    "postal_code": "10100",
    "client_phone": "62123412341234",
    "createdAt": "2024-01-01T00:00:00.000Z",
    "updatedAt": "2024-01-02T00:00:00.000Z"
  }
}
```

Kode 3.82: Response 200 - Update Berhasil

- **Error Response**
  - **Code:** `400 Bad Request`
    Deskripsi: Parameter tidak valid

```
{
  "status": "error",
  "code": 400,
  "message": "Invalid parameters",
  "data": null
}
```

Kode 3.83: Response 400 - Parameter Tidak Valid

  - **Code:** `404 Not Found`
    Deskripsi: Client tidak ditemukan

```
{
  "status": "error",
  "code": 404,
  "message": "Client not found",
  "data": null
}
```

Kode 3.84: Response 404 - Client Tidak Ditemukan

- **Code:** `500 Internal Server Error`
  Deskripsi: Terjadi kesalahan pada server

```
{
  "status": "error",
  "code": 500,
  "message": "Internal server error",
  "data": null
}
```

Kode 3.85: Response 500 - Server Error

3.6.13 POST /invoices

- **Tags:** `Invoices`
- **Summary:** Membuat *invoice* baru beserta detailnya
- **URL Params**
  - Tidak ada URL parameter
- **Headers**
  - `Content-Type: application/json`
  - `cookies: accessToken`
- **Request Body**

```
{
  "invoice_number": "INV-2024-001",
  "issue_date": "2024-05-01",
  "due_date": "2024-05-31",
  "tax_rate": 0.1,
  "tax_invoice_number": "TAX-INV-001",
  "client_id": "123e4567-e89b-12d3-a456-426614174000",
```

```
  "invoice_details": [
    {
      "transaction_note": "Pengiriman bulan Mei",
      "delivery_count": 5,
      "price_per_delivery": 30000
    }
  ]
}
```

Kode 3.86: Request Body - Buat Invoice Baru

- **Success Response**
  - **Code:** `201 Created`
  - **Deskripsi:** Invoice berhasil dibuat

```
{
  "status": "success",
  "code": 201,
  "message": "Successfully created invoice",
  "data": {
    "invoice_id": "987e6543-e21b-12d3-a456-426614174000",
    "invoice_number": "INV-2024-001",
    "issue_date": "2024-05-01T00:00:00.000Z",
    "due_date": "2024-05-31T00:00:00.000Z",
    "sub_total": 150000,
    "tax_rate": 0.1,
    "tax_amount": 15000,
    "total": 165000,
    "tax_invoice_number": "TAX-INV-001",
    "amount_paid": 0,
    "payment_status": "unpaid",
    "voided_at": null,
    "client_id": "123e4567-e89b-12d3-a456-426614174000",
    "created_at": "2024-05-01T10:00:00.000Z",
    "updated_at": "2024-05-01T10:00:00.000Z"
  }
}
```

Kode 3.87: Response 201 - Invoice Berhasil Dibuat

NUSANTARA

- **Error Response**
  - **Code:** `400 Bad Request`
    Deskripsi: Parameter tidak valid

```
{
  "status": "error",
  "code": 400,
  "message": "Invalid parameters",
  "data": {
    "invoice_number": ["Invoice number is required"]
  }
}
```

Kode 3.88: Response 400 - Parameter Tidak Valid

- **Code:** `401 Unauthorized`
  Deskripsi: Token tidak valid atau belum *login*

```
{
  "status": "error",
  "message": "Authentication required",
  "data": null
}
```

Kode 3.89: Response 401 - Tidak Terautentikasi

- **Code:** `403 Forbidden`
  Deskripsi: Akses tidak diizinkan

```
{
  "status": "error",
  "message": "Forbidden, role not sufficient",
  "data": null
}
```

Kode 3.90: Response 403 - Akses Ditolak

- **Code:** `404 Not Found`
  Deskripsi: Client tidak ditemukan

```
{
  "status": "error",
  "code": 404,
  "message": "Client not found",
  "data": null
}
```

Kode 3.91: Response 404 - Client Tidak Ditemukan

- **Code:** `409 Conflict`
  Deskripsi: Invoice dengan nomor yang sama sudah ada

```
{
  "status": "error",
```

```
  "code": 409,
  "message": "Duplicate invoice",
  "data": null
}
```

Kode 3.92: Response 409 - Konflik Duplikasi

- **Code:** `500 Internal Server Error`
  Deskripsi: Terjadi kesalahan pada server

```
{
  "status": "error",
  "code": 500,
  "message": "Internal server error",
  "data": null
}
```

Kode 3.93: Response 500 - Server Error

3.6.14 GET /invoices

- **Tags:** `Invoices`
- **Summary:** Mengambil semua data *invoice* beserta detail transaksi dan data klien
- **URL Params**
  - Tidak ada URL parameter
- **Headers**
  - `cookies: accessToken`
  - `Content-Type: application/json`
- **Request Body**
  - Tidak ada body yang dibutuhkan
- **Success Response**
  - **Code:** `200 OK`
  - **Deskripsi:** Data invoice berhasil diambil

```
{
  "status": "success",
  "code": 200,
  "message": "Data successfully retrieved",
  "data": [
    {
      "invoice_id": "987e6543-e21b-12d3-a456-426614174000",
      "invoice_number": "INV-2024-001",
      "issue_date": "2024-05-01T00:00:00.000Z",
      "due_date": "2024-05-31T00:00:00.000Z",
      "tax_rate": 0.1,
      "tax_amount": 15000,
      "sub_total": 150000,
      "total": 165000,
      "tax_invoice_number": "TAX-INV-001",
      "amount_paid": 0,
      "payment_status": "unpaid",
      "voided_at": null,
      "client_id": "123e4567-e89b-12d3-a456-426614174000",
      "created_at": "2024-05-01T10:00:00.000Z",
      "updated_at": "2024-05-01T10:00:00.000Z",
      "client": {
        "client_id": "123e4567-e89b-12d3-a456-426614174000",
        "name": "PT Maju Jaya",
        "address": "Jl. Contoh No.123",
        "phone": "08123456789",
        "created_at": "2024-01-01T00:00:00.000Z",
        "updated_at": "2024-01-02T00:00:00.000Z"
      },
      "invoice_details": [
        {
          "invoice_detail_id": "abc123-e89b-12d3-a456-426614174001",
          "transaction_note": "Pengiriman barang X",
          "delivery_count": 10,
          "price_per_delivery": 15000,
          "amount": 150000,
          "invoice_id": "987e6543-e21b-12d3-a456-426614174000",
          "createdAt": "2024-05-01T10:00:00.000Z",
```

```
            "updatedAt": "2024-05-01T10:00:00.000Z"
          }
        ]
      }
    ]
}
```

Kode 3.94: Response 200 - Ambil Semua Invoice

- **Error Response**
  - **Code:** `500 Internal Server Error`
    Deskripsi: Terjadi kesalahan internal pada server

```
{
  "status": "error",
  "code": 500,
  "message": "Internal server error",
  "data": null
}
```

Kode 3.95: Response 500 - Server Error

3.6.15 GET /invoices/{id}

- **Tags:** `Invoices`
- **Summary:** Mengambil satu data *invoice* lengkap dengan detail dan data *client* berdasarkan `id`
- **URL Params**
  - `id [UUID] (required)` — UUID dari invoice yang ingin diambil
- **Headers**
  - `cookies: accessToken`
  - `Content-Type: application/json`
- **Request Body**
  - Tidak ada body yang dibutuhkan
- **Success Response**

- **Code:** `200 OK`
- **Deskripsi:** Invoice berhasil diambil

```
{
  "status": "success",
  "code": 200,
  "message": "Data successfully retrieved",
  "data": {
    // Struktur sesuai dengan InvoiceWithClientAndDetails
    // (lihat definisi pada /invoices GET)
  }
}
```

Kode 3.96: Response 200 - Invoice Ditemukan

- **Error Response**
  - **Code:** `400 Bad Request`
    Deskripsi: Parameter *invoice* `id` tidak valid atau tidak diberikan

```
{
  "status": "error",
  "code": 400,
  "message": "Invalid parameters",
  "data": {
    "message": "Invoice ID is required"
  }
}
```

Kode 3.97: Response 400 - Parameter Tidak Valid

  - **Code:** `404 Not Found`
    Deskripsi: *Invoice* dengan `id` yang diberikan tidak ditemukan

```
{
  "status": "error",
  "code": 404,
  "message": "Invoice not found"
}
```

Kode 3.98: Response 404 - Invoice Tidak Ditemukan

  - **Code:** `500 Internal Server Error`
    Deskripsi: Terjadi kesalahan internal pada server

```
{
  "status": "error",
  "code": 500,
```

```
  "message": "Internal server error",
  "data": null
}
```

Kode 3.99: Response 500 - Server Error

3.6.16 PUT /invoices/{id}

- **Tags:** `Invoices`
- **Summary:** Memperbarui data *invoice* berdasarkan `id` beserta detail transaksinya
- **URL Params**
  - `id [UUID] (required)` — UUID dari *invoice* yang ingin diperbarui
- **Headers**
  - `cookies: accessToken`
  - `Content-Type: application/json`
- **Request Body**

```
{
  "invoice_number": "INV-2024-001",
  "issue_date": "2024-05-01",
  "due_date": "2024-05-31",
  "tax_rate": 0.1,
  "tax_invoice_number": "TAX-INV-001",
  "voided_at": null,
  "client_id": "123e4567-e89b-12d3-a456-426614174000",
  "invoice_details": [
    {
      "invoice_detail_id": "abc123-e89b-12d3-a456
    -426614174001",
      "transaction_note": "Pengiriman barang X",
      "delivery_count": 10,
      "price_per_delivery": 15000,
      "invoice_id": "987e6543-e21b-12d3-a456-426614174000"
    }
  ]
}
```

Kode 3.100: Request Body - Update Invoice

- **Success Response**
  - **Code:** `200 OK`
  - **Deskripsi:** Data invoice berhasil diperbarui

```
{
  "status": "success",
  "code": 200,
  "message": "Data successfully updated",
  "data": {
    // Struktur data invoice setelah diperbarui
  }
}
```

Kode 3.101: Response 200 - Invoice Diperbarui

- **Error Response**
  - **Code:** `400 Bad Request`
    Deskripsi: Parameter tidak valid atau ada field yang hilang

```
{
  "status": "error",
  "code": 400,
  "message": "Invalid parameters",
  "data": {
    "field": "client_id",
    "message": "Client ID is required"
  }
}
```

Kode 3.102: Response 400 - Parameter Tidak Valid

  - **Code:** `404 Not Found`
    Deskripsi: *Invoice* atau *client* tidak ditemukan

```
{
  "status": "error",
  "code": 404,
  "message": "Invoice not found"
}
```

Kode 3.103: Response 404 - Tidak Ditemukan

  - **Code:** `500 Internal Server Error`
    Deskripsi: Terjadi kesalahan internal pada server

```
{
  "status": "error",
  "code": 500,
  "message": "Internal server error",
  "data": null
}
```

Kode 3.104: Response 500 - Server Error

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:

1. Terdapat kurangnya komunikasi yang jelas terkait desain antarmuka pengguna (*user interface*), khususnya mengenai alur pertukaran data antara *frontend* dan *backend*. Hal ini mengakibatkan miskomunikasi antar tim dan menyebabkan ketidaksesuaian dalam implementasi fungsionalitas.

2. Karena sistem yang dikembangkan merupakan sistem keuangan (*finance system*), terdapat tantangan dalam memahami istilah-istilah teknis dan alur bisnis yang digunakan oleh tim keuangan. Kurangnya pemahaman terhadap proses pembuatan *invoice* dan prosedur kerja yang dijalankan oleh tim *finance* menjadi salah satu faktor yang memperlambat proses pengembangan.

3. Terdapat situasi ketika salah satu anggota tim sempat meninggalkan pekerjaannya yang belum terselesaikan tanpa pemberitahuan dan tanpa dokumentasi yang memadai. Hal ini sempat menghambat kelancaran proses pengembangan, karena anggota tim lain harus memahami terlebih dahulu konteks dari pekerjaan tersebut. Namun, pada tahap akhir proyek, yang bersangkutan kembali bergabung dan memberikan klarifikasi terkait tugas yang sebelumnya belum selesai, sehingga permasalahan tersebut dapat diselesaikan dengan lebih cepat.

3.7.2 Solusi

1. Untuk mengatasi kurangnya komunikasi yang jelas terkait desain antarmuka pengguna, khususnya mengenai alur pertukaran data antara *frontend* dan *backend*, tim mulai menerapkan dokumentasi teknis yang lebih sistematis dan terstandarisasi. Seluruh perubahan dan keputusan terkait desain disampaikan melalui pertemuan rutin harian (*daily stand-up*) dan dicatat dalam dokumen kolaboratif. Selain itu, pembuatan skema API serta *wireframe* ditetapkan sebagai langkah awal wajib sebelum implementasi dilakukan, guna menyamakan persepsi antar tim.

2. Mengingat sistem yang dikembangkan berkaitan dengan proses keuangan, terdapat kendala dalam memahami istilah teknis dan alur bisnis yang belum sepenuhnya dipahami oleh tim pengembang. Untuk mengatasinya, dilakukan sesi tanya jawab secara langsung dengan *supervisor* sebagai narasumber utama. *Supervisor* memberikan penjelasan mendalam mengenai proses bisnis, pembuatan *invoice*, serta prosedur keuangan yang berlaku di perusahaan. Pendekatan ini membantu tim dalam menerjemahkan kebutuhan bisnis ke dalam bentuk fungsionalitas sistem yang sesuai.

3. Untuk mengantisipasi terhambatnya pengembangan akibat ketidakhadiran atau pengunduran diri anggota tim, diterapkan kebijakan dokumentasi wajib pada setiap tugas yang sedang dikerjakan. Dokumentasi ini mencakup penjelasan fungsi, alur logika, serta status pekerjaan terakhir. Meski sempat terjadi kendala ketika salah satu anggota tim meninggalkan tugas tanpa dokumentasi, kembalinya yang bersangkutan di tahap akhir proyek membantu mempercepat penyelesaian. Kejadian ini menjadi bahan evaluasi penting dalam memperkuat budaya kerja kolaboratif dan tanggung jawab tim.

UNIVERSITAS MULTIMEDIA NUSANTARA