BAB 3
PELAKSANAAN KERJA MAGANG

3.1 Kedudukan dan Koordinasi

Selama pelaksanaan kegiatan kerja praktek di PT Cranium Royal Aditama, penulis ditempatkan dalam divisi *Product Development*, khususnya pada posisi *Full Stack Developer* dalam proyek pengembangan perangkat lunak *Enterprise Resource Planning* (ERP) milik perusahaan. Fokus utama selama periode magang adalah pada proses pengembangan serta penyempurnaan modul-modul yang terdapat dalam sistem ERP yang akan ditawarkan Cranium kepada para klien korporat.

Dalam aktivitas sehari-hari, penulis berada di bawah arahan beberapa mentor dan *supervisor* yang memberikan bimbingan teknis, pemantauan terhadap progres pekerjaan, serta evaluasi terhadap hasil tugas yang telah diselesaikan. Selain itu, penulis juga berkolaborasi dengan anggota tim pengembangan ERP lainnya, di mana masing-masing individu memiliki tanggung jawab spesifik dalam pengembangan sistem ERP

Struktur kedudukan dalam proyek ERP Cranium dapat dijabarkan sebagai berikut:

1. *Supervisor*

   Terdapat dua sosok utama yang berperan sebagai pengawas dalam proyek ini. Yang pertama adalah Bapak Sugito selaku *VP of Engineering* sekaligus *Tech Lead*, dan yang kedua adalah Bapak Angga Tama sebagai *Project Manager* untuk sistem ERP. Keduanya memiliki tanggung jawab utama dalam mengarahkan pekerjaan para *developer*, memastikan penyelesaian tugas secara tepat waktu, serta memverifikasi kesesuaian hasil kerja dengan standar perusahaan. Selain itu, mereka turut memberikan pengenalan terkait budaya kerja, prosedur teknis, serta ekspektasi yang harus dipenuhi oleh mahasiswa magang di Cranium.

2. Mentor

   Dalam melaksanakan tugas sebagai *full stack developer*, penulis didampingi oleh dua mentor yang masing-masing membimbing pada aspek *back-end* dan *front-end* pengembangan ERP. Para mentor memberikan arahan teknis mulai dari tahap perkenalan lingkungan kerja, pengenalan struktur kode, hingga praktik kolaborasi tim melalui penggunaan sistem kontrol versi seperti

GitHub. Para mentor juga secara langsung mengawasi dan memastikan hasil kerja para Mahasiswa Magang agar selaras dengan standar perusahaan, sebelum diperiksa kembali oleh *supervisor*.

3. Mahasiswa Magang
   Penulis sebagai peserta magang memiliki tanggung jawab untuk menyelesaikan tugas-tugas yang telah ditentukan oleh *supervisor*, sesuai dengan tenggat waktu dan spesifikasi teknis yang ditetapkan. Beberapa tugas utama yang dikerjakan antara lain adalah menambahkan sistem validasi pada sistem, menambahkan fitur *anti double* input, penghapusan fitur, dan perbaikan *bugs*.

3.2 Alur dan Prosedur Kerja Proyek

Selama pelaksanaan kerja magang, terdapat sejumlah peraturan dan prosedur yang ditetapkan oleh PT Cranium Royal Aditama untuk memastikan pengembangan proyek ERP dapat berjalan secara efektif dan efisien. Prosedur ini dirancang untuk menjaga koordinasi yang baik antar anggota tim, serta menjamin bahwa setiap fitur dikembangkan sesuai standar yang ditentukan. Berikut adalah alur kerja yang diterapkan dalam proyek ERP Cranium:

1. Distribusi Tugas
   (a) Tugas baru diberikan setelah *developer* menyelesaikan tugas sebelumnya secara penuh, termasuk tahap penggabungan *(merge)* ke dalam *development branch* pada *repository*.
   (b) Tugas akan diberikan secara langsung oleh *supervisor* kepada *developer*, dan dikomunikasikan melalui roadmap pengerjaan ERP. Detail teknis pengerjaan suatu tugas akan di-*brief* secara langsung oleh *supervisor* secara pribadi maupun dalam *meeting* dengan keseluruhan tim ERP Cranium.
   (c) Tugas dapat berupa pengemangan tampilan antarmuka *(front-end)*, pengembangan logika sistem *(back-end)*, maupun perbaikan *bugs* atau modifikasi fitur yang sudah ada.
   (d) Setiap tugas dikerjakan pada *branch* yang berbeda dalam *repository*. Hal ini dilakukan untuk mempermudah integrasi hasil kerja antar

developer dan mengantisipasi terjadinya *conflict* antar hasil kerja *developer*.

2. Pengerjaan Tugas

   (a) Tugas yang sudah di-*brief* oleh *supervisor* akan langsung dikerjakan oleh para *developer*. Tugas akan dikerjakan sesuai dengan arahan *supervisor* dan mengikuti standar pengerjaan dalam perusahaan dalam jangka waktu yang sudah ditentukan oleh *supervisor*.

   (b) Mahasiswa magang diminta untuk mengerjakan tugasnya secara maksimal dan mandiri, namun apabila mengalami kendala dalam pengerjaan, maka kedua mentor akan turut serta membantu mencari solusi terhadap kendala yang ditemukan.

3. Pengujian Hasil Kerja

   Setiap hasil kerja mahasiswa magang diuji dan divalidasi sebelum dilakukan *pull request* dan *merge* ke *branch development* proyek ERP Cranium. Proses pengujian dilakukan dengan dua metode:

   (a) *Unit Test* dan *Contract Test*

   Pengujian otomatis dilakukan dengan fitur *testing framework Spring Boot* untuk *back-end* dan *Next.js* untuk *front-end. Test case* disesuaikan dengan fitur yang ada pada ERP Cranium.

   (b) Pengujian Manual

   Pengujian dilakukan langsung melalui tampilan antarmuka pengguna untuk memastikan fitur berjalan sesuai kebutuhan dan tidak menyebabkan *bug* atau *conflict* sistem.

4. Konsultasi dan Revisi

   Setelah tahap pengujian, hasil kerja mahasiswa magang akan diperiksa oleh mentor melalui proses *pull request*. Pemeriksaan difokuskan pada struktur kode dan kesesuaian dengan standar penulisan yang digunakan dalam pengembangan ERP Cranium, serta memastikan bahwa tidak terdapat *conflict* dengan bagian kode lain pada *branch development*. Apabila terdapat revisi atau masukan terhadap hasil kerja, maka para mentor akan menandakan kesalahan dan memberi *comment* pada *pull request*. Mahasiswa magang kemudian wajib melakukan revisi sesuai arahan tersebut dalam waktu yang telah disepakati. Jika terdapat kendala dalam pengerjaan, mahasiswa dapat

berdiskusi kembali untuk mendapatkan solusi atau tambahan waktu. Apabila revisi selesai dikerjakan, hasil kerja kan kembali diperiksa oleh mentor untuk memastikan kesesuaian hasil revisi.

5. Penggabungan Hasil Kerja
   Setelah hasil kerja mahasiswa magang dinyatakan telah sesuai oleh mentor dan revisi yang diminta telah diselesaikan, tahap berikutnya adalah proses penggabungan kode atau *merge* ke dalam *branch Development* Cranium ERP. Penggabungan dilakukan oleh mentor setelah memastikan bahwa kode sudah sesuai standar dan tidak menimbulkan konflik dengan bagian lain dalam sistem. Tahap ini menandai bahwa fitur atau perbaikan yang dikerjakan mahasiswa telah menjadi bagian dari *branch development* Cranium ERP dan siap digunakan lebih lanjut oleh *developer* lainnya.

6. Finalisasi Hasil Kerja
   Hasil kerja dianggap final apabila sudah digabungkan dalam *branch development* Cranium ERP. Setelah digabungkan, branch yang digunakan untuk pengembangan fitur dihapus dari *repository* dan *developer* membuat *branch* baru untuk pengerjaan tugas berikutnya.

Alur komunikasi dan koordinasi selama kerja magang di PT Cranium Royal Aditam dirancang agar setiap anggota tim dapat bekerja secara efektif dan terarah. Penjabaran mengenai komunikasi, koordinasi, dan kolaborasi selama proyek ERP Cranium adalah sebagai berikut:

1. Komunikasi
   (a) Komunikasi harian dalam proyek ERP dilakukan melalui aplikasi *WhatsApp* sebagai sarana utama dalam menyampaikan informasi, diskusi tugas, serta pengingat *deadline*.
   (b) Aplikasi *Discord* digunakan sebagai media komunikasi tambahan, terutama ketika bekerja dari rumah (WFH), untuk diskusi langsung, konsultasi teknis, atau koordinasi kelompok.
   (c) Dalam kondisi WFO (*Work From Office*), komunikasi dilakukan secara langsung melalui pertemuan informal, diskusi pribadi, maupun rapat tim di kantor.

2. Koordinasi

(a) Koordinasi *progress* kerja dilakukan menggunakan tabel *roadmap* yang dibuat di *Figma*, yang berisi daftar tugas, status pengerjaan, dan pelacakan waktu penyelesaian.

(b) Mahasiswa magang turut berperan aktif dalam memperbarui status tugas pada *roadmap Figma*, sebagaimana dilakukan oleh anggota tim lainnya.

(c) *Google Sheets* kolaboratif disediakan untuk mencatat hasil kerja harian, memungkinkan anggota tim untuk saling memantau perkembangan masing-masing.

3. Kolaborasi

(a) Kolaborasi teknis dalam pengerjaan proyek ERP dilakukan melalui *GitHub*, yang digunakan sebagai repositori utama proyek.

(b) Setiap *developer*, termasuk mahasiswa magang, mengerjakan tugas pada *branch* yang terpisah agar mempermudah proses integrasi dan menjaga kestabilan proyek.

3.3 Tugas yang Dilakukan

Selama masa magang di PT Cranium Royal Aditama, penulis berkontribusi dalam proyek pengembangan sistem *Enterprise Resource Planning* (ERP) yang dimiliki oleh perusahaan. Tugas utama yang diberikan meliputi pengembangan dan penyempurnaan berbagai fitur dalam sistem ERP, disesuaikan dengan kebutuhan internal perusahaan maupun permintaan dari klien.

Pengembangan sistem ERP Cranium dilakukan dengan menerapkan pendekatan arsitektur modular monolitik, di mana bagian *back-end* dikembangkan menggunakan *Java Spring Boot*, sementara bagian *front-end* menggunakan *Next.js* dengan *TypeScript*. Arsitektur modular monolitik menggabungkan kelebihan sistem monolitik dalam hal kemudahan *deployment* dengan prinsip modularitas yang umum dijumpai pada arsitektur *microservices*. Dalam pendekatan arsitektur ini, sistem dipecah menjadi sejumlah modul yang bersifat independen satu sama lain, di mana masing-masing modul memiliki tanggung jawab yang spesifik serta hubungan ketergantungan (*dependency*) yang telah ditentukan secara jelas. Tiap modul disusun untuk menjalankan fungsi yang berbeda sesuai dengan cakupan kerjanya, dan dilengkapi dengan antarmuka (*interface*) yang terdefinisi dengan jelas untuk mendukung integrasi antar modul. Setiap modul dirancang untuk menangani fungsi

tertentu, dilengkapi dengan antarmuka yang eksplisit, serta memungkinkan proses pengembangan dan pengujian dilakukan secara terpisah.

Jenis tugas yang ditangani selama masa magang sangat beragam, mencakup perbaikan *bugs*, pengembangan fitur pada sisi *front-end* maupun *back-end*, serta penyesuaian terhadap fitur yang sudah ada. Setiap tugas diberikan berdasarkan pembagian kerja yang ditentukan oleh *supervisor* dan dikerjakan sesuai dengan standar teknis serta kebutuhan aktual dari proyek. Melalui berbagai tugas tersebut, penulis dapat terlibat langsung dalam proses pengembangan perangkat lunak secara menyeluruh dan membangun pemahaman yang kuat mengenai praktik kerja profesional dalam pengembangan sistem, sekaligus menyesuaikan diri dengan kebutuhan perusahaan.

Dalam menjalankan tugas-tugas tersebut, digunakan berbagai *tools, software,* dan *framework* yang mendukung produktivitas dan efisiensi kerja. Penjelasan mengenai *tools, framework,* dan *software* yang digunakan dalam pengembangan ERP Cranium serta penjelasannya adalah sebagai berikut:

1. *IntelliJ IDEA*
   Digunakan untuk mengembangkan *back-end* berbasis *Java Spring Boot*. IDE ini terintegrasi dengan baik dengan *Java*, sehingga mempermudah proses pengembangan, *debugging*, serta pembuatan *unit test* dan *contract test*.

2. *Java Spring Boot*
   Berperan sebagai *framework* utama dalam pengembangan *back-end* ERP. *Framework* ini mendukung pembuatan aplikasi berbasis *Java* secara *modular*, cepat, dan terstruktur.

3. *Visual Studio Code*(VSCode)
   Digunakan untuk mengembangkan *front-end* ERP yang dibangun menggunakan *Next.js* dan *TypeScript*. VSCode juga digunakan dalam pembuatan *unit test* untuk *front-end*.

4. *Next.js* dengan *TypeScript*
   Digunakan sebagai *framework front-end* untuk membangun antarmuka pengguna ERP dalam bentuk *web application*.

5. *PostgreSQL*
   Digunakan sebagai sistem manajemen basis data relasional untuk menyimpan seluruh data dalam proyek ERP Cranium.

6. *DBeaver*

   Digunakan sebagai *database client* GUI untuk mengelola dan menavigasi *database PostgreSQL* secara efisien selama proses pengembangan.

7. *Postman*

   Digunakan sebagai *tool* untuk menguji dan memverifikasi API yang dikembangkan pada sisi *back-end* agar dapat digunakan dengan baik oleh *front-end*.

8. *Git*

   Digunakan sebagai *version control system* untuk melacak dan mengelola perubahan pada kode sumber selama proyek berjalan.

9. *GitHub*

   Digunakan sebagai platform kolaborasi dan repositori untuk menyimpan, mengelola, serta melakukan *code review* terhadap proyek ERP secara terdistribusi.

3.4 Uraian Pelaksanaan Magang

3.4.1 Pelaksanaan Kerja Magang

Selama menjalani masa magang selama 20 minggu, penulis telah menyelesaikan berbagai jenis pekerjaan yang mencakup beragam aspek pengembangan ERP Cranium. Seluruh aktivitas tersebut dilakukan secara berkelanjutan dan mencerminkan kontribusi dalam pengembangan ERP Cranium. Berikut ini merupakan rangkuman pekerjaan yang telah dilaksanakan penulis selama periode magang, yang dapat dilihat pada Tabel 3.1.

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

Minggu Ke -	Pekerjaan yang dilakukan
1	Instalasi *software* pendukung, Perkenalan perusahaan beserta SOP dalam mengerjakan *project*.
2	Eksplorasi dan *training* mengenai sisi *back-end* perusahaan menggunakan *Java Spring Boot*. Meliputi pembelajaran konsep dari DTO, *service*, *controller*, dan *repository*.
	Lanjut di halaman berikutnya

Tabel 3.1 – lanjutan dari halaman sebelumnya

Minggu Ke -	Pekerjaan yang dilakukan
3	Praktik sisi *back-end* dengan membuat entitas *custom*, serta pembelajaran lanjutan mengenai *auth* dan *scope*.
4	*Training* mengenai *contract test* dan *unit test*, beserta mengaplikasikannya ke entitas *custom*.
5	Hasil dari *training back-end* diulas dan dievaluasi oleh *supervisor*
6	Eksplorasi dan *training* sisi *front-end* perusahaan menggunakan *NextJS*.
7	*Training* mengenai konsep dasar CRUD beserta praktik pembuatan halaman *create* dan *read* (*POST* dan *GET*).
8	*Training front-end* yang berfokus pada pembuatan halaman *update* dan *delete* , serta pembelajaran tentang *unit test front-end*.
9	Lanjutan *training unit test front-end* serta evaluasi hasil *training* oleh *supervisor*.
10	persiapan *cloning repository development* ERP Cranium, beserta eksplorasi ERP Cranium.
11	Melakukan fix *bug* untuk modul *Master*, di submodul *Chart of Account.*
12	Menambahkan Validasi pada kolom nama pada *Master Warehouse* agar tidak terjadi duplikat data.
13	Menambahkan Validasi pada kolom nama pada *Master Supplier* agar tidak terjadi duplikat data.
14	Menambahkan Validasi pada kolom nama pada *Master Sales* agar tidak terjadi duplikat data.
15	Menambahkan Validasi pada kolom nomor kendaraan *VehicleNo* pada *Master Vehicle* agar tidak terjadi duplikat data.
16	Menambahkan Validasi pada kolom nama pada *Master Department* agar tidak terjadi duplikat data.
17	Menambahkan fitur *Order fulfillment* pada modul *selling* yang mencegah duplikasi proses retur terhadap dokumen yang sama.

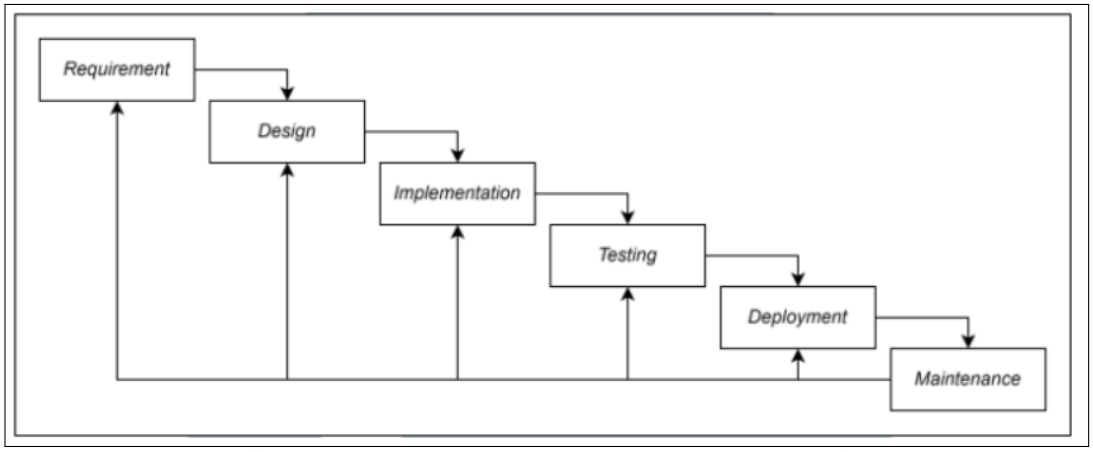
Lanjut di halaman berikutnya

Tabel 3.1 – lanjutan dari halaman sebelumnya

Minggu Ke -	Pekerjaan yang dilakukan
18	Lanjutan penambahan fitur *Order fulfillment* pada modul *selling*.
19	Penghapusan fitur *Create By Reference* saat melakukan *create* pada submodul *Item Withdraw* di modul *Warehouse*.
20	*Bug fixing* pada *Audit trail* di Modul *Warehouse*, meliputi submodul *Item Withdraw, Stock Opname, Item Receipt,* dan *Item Transfer Reservation*.

3.4.2 Konsep *Sofware Development Life Cycle* ERP Cranium

Software Development Life Cycle (SDLC) atau Siklus Hidup Pengembangan Perangkat Lunak merupakan suatu kerangka kerja sistematis yang digunakan untuk menggambarkan tahapan-tahapan dalam proses pengembangan, pengelolaan, dan pemeliharaan perangkat lunak. Model ini bertujuan untuk memastikan bahwa perangkat lunak dikembangkan secara terstruktur dan efisien sesuai dengan kebutuhan pengguna dan tujuan bisnis. Ilustrasi alur SDLC ERP Cranium dapat dilihat pada Gambar 3.1.



Gambar 3.1. Alur SDLC ERP Cranium

Dalam pengembangan perangkat lunak ERP Cranium, pendekatan SDLC yang diterapkan adalah *Waterfall Model*, yaitu model pengembangan perangkat lunak yang bersifat sekuensial dan linear. Pada model ini, setiap tahapan harus diselesaikan secara menyeluruh sebelum melanjutkan ke tahap berikutnya.

Alur pengerjaan mengalir satu arah dari tahap awal hingga tahap akhir, dengan dokumentasi dan persetujuan di setiap fase.

Berikut adalah penjelasan masing-masing tahap dalam model *Waterfall* sebagaimana diterapkan dalam proyek ERP Cranium:

- *Requirement:*

  Tahapan awal ini difokuskan pada pengumpulan dan analisis kebutuhan sistem, baik dari sisi teknis maupun bisnis. Aktivitas pada tahap ini mencakup identifikasi kebutuhan perangkat lunak, penyusunan spesifikasi fungsional, serta pemahaman terhadap proses bisnis yang akan didukung oleh sistem ERP.

- *Design:*

  Berdasarkan hasil analisis kebutuhan, tahap perancangan dilakukan untuk menentukan arsitektur sistem secara menyeluruh. Desain mencakup struktur modul, interaksi antar komponen, rancangan basis data, serta antarmuka pengguna (*UI*). Dokumen desain ini menjadi acuan utama dalam proses implementasi.

- *Implementation:*

  Pada tahap ini, tim pengembang mulai menulis kode program berdasarkan spesifikasi desain yang telah disetujui. Pengembangan dilakukan secara modular, dimulai dari pembuatan fungsionalitas dasar seperti proses CRUD (*Create, Read, Update, Delete*) untuk berbagai entitas dalam sistem. Tahap ini juga mencakup penulisan *business logic* dan integrasi dasar antar modul.

- *Testing:*

  Setelah implementasi selesai, dilakukan tahap pengujian untuk memverifikasi bahwa seluruh fungsionalitas berjalan sesuai dengan desain dan tidak terdapat bugs yang dapat mempengaruhi kinerja sistem. Pengujian dilakukan baik secara manual maupun dengan menggunakan *unit test* untuk memastikan kualitas perangkat lunak.

- *Deployment:*

  Apabila sistem telah lolos tahap pengujian, maka dilanjutkan dengan proses *deployment* ke lingkungan produksi. Tahap ini mencakup instalasi *software*

ke *server* dan aktivitas lainnya untuk memastikan aplikasi dapat berjalan dengan baik dan dapat digunakan oleh user.

- *Maintenance:*

  Setelah sistem digunakan, proses pemeliharaan dilakukan secara berkala. Tahap ini mencakup perbaikan *bug* yang ditemukan pasca-*deployment*, penyempurnaan kinerja sistem, serta penambahan fitur baru untuk menjawab kebutuhan pengguna yang berkembang dari waktu ke waktu.

3.4.3 Ruang Lingkup dan Konsep Pengembangan

Pengembangan yang dilakukan selama kerja magang pada proyek ERP Cranium mencakup beberapa ruang lingkup penting yang dibagi menjadi poin-poin utama berikut:

1. Pengembangan Fitur

   Pengembangan fitur dilakukan untuk menambahkan kapabilitas baru pada sistem ERP Cranium agar dapat memenuhi kebutuhan bisnis pengguna. Pengembangan mencakup Pencegahan *double input*, penambahan validasi serta penyesuaian fungsionalitas sesuai dengan alur bisnis perusahaan.

2. Perbaikan *Bugs*

   Perbaikan bugs merupakan bagian penting dari pengembangan, mencakup bugs yang ditemukan selama pengerjaan maupun yang telah dilaporkan sebelumnya. Perbaikan dilakukan pada sisi *front-end* maupun *back-end*, dengan cakupan sebagai berikut:

   (a) Perbaikan tampilan data pada *web*

   Mencakup penyesuaian label dan *translation* pada kolom atau *field* agar sesuai dengan konteks informasi yang ditampilkan. Selain itu, dilakukan penyesuaian terhadap struktur data yang dikirim dan diterima agar sesuai dengan ekspektasi sistem. Pembaruan ini bertujuan untuk memastikan bahwa data tampil secara utuh, akurat, dan mudah dipahami oleh pengguna.

   (b) Perbaikan struktur data *front-end* dan *back-end*

Memastikan kesesuaian format data antara sisi *front-end* dan *back-end*, sehingga pertukaran data antar komponen dapat berlangsung dengan lancar dan tanpa *error*.

Seluruh ruang lingkup tersebut diselesaikan dengan pendekatan bertahap dan berorientasi pada kebutuhan sistem, sehingga ERP Cranium dapat berfungsi secara optimal, akurat, dan andal dalam mendukung operasional perusahaan.

Dalam proses pengembangan ERP Cranium, terdapat sejumlah konsep inti yang secara konsisten digunakan sebagai dasar dalam membangun struktur dan logika sistem. Berikut adalah uraian konsep yang umum digunakan dalam pengembangan ERP Cranium :

1. *Data Transfer Object, Entity, & Mapper*

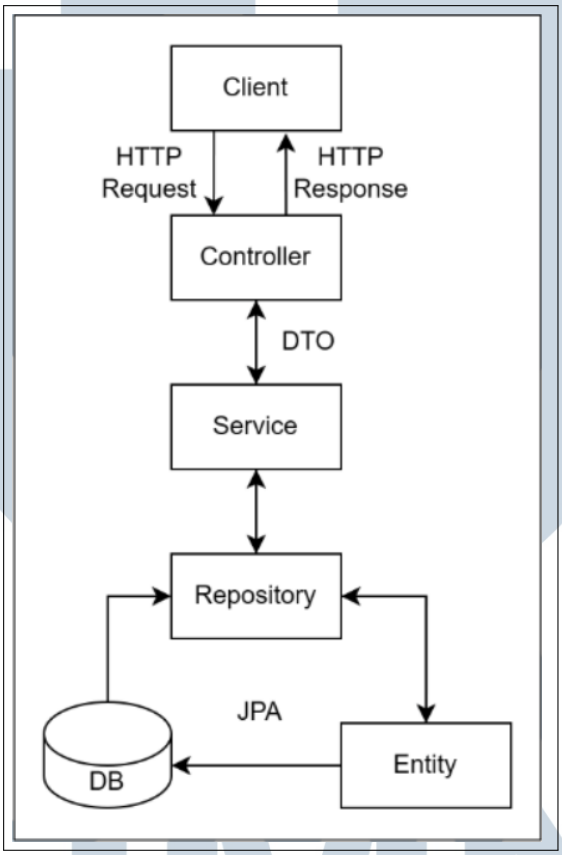
   Dalam pengembangan sistem ERP Cranium, digunakan konsep pemisahan data melalui penggunaan *Data Transfer Object* (DTO), *entity*, dan *mapper*. DTO merupakan objek sederhana yang berfungsi membawa data dari satu lapisan ke lapisan lain tanpa mengandung logika bisnis. Objek ini dirancang agar efisien untuk komunikasi antar proses, seperti antara *client* dan *server*, karena hanya memuat data yang dibutuhkan dan menyaring informasi yang bersifat *internal*, seperti struktur tabel database. Hal ini mencegah kebocoran data sensitif sekaligus menyederhanakan komunikasi antar bagian sistem.

   Entity merupakan representasi dalam bentuk objek Java yang mencerminkan struktur tabel pada database relasional. *Entity* digunakan sebagai objek persistensi dan berisi anotasi dari *Java Persistence API* (JPA) atau *Jakarta Persistence* untuk mengatur relasi antar entitas, nama kolom, dan atribut database lainnya. *Entity* menjadi komponen penting dalam ORM (*Object-Relational Mapping*) karena memungkinkan pengembang mengakses dan memanipulasi data dalam bentuk objek, bukan *query SQL* mentah.

   Untuk menjembatani antara DTO dan *entity*, digunakan komponen *mapper*. *Mapper* bertanggung jawab melakukan konversi dua arah: dari DTO ke *entity* untuk proses penyimpanan ke database, dan dari *entity* ke DTO untuk keperluan penyajian data ke *front-end*. Proses ini memastikan konsistensi dan keamanan data sepanjang alur pemrosesan dalam aplikasi.

2. *Controller, Service, & Repository*

Struktur pengembangan *back-end* ERP Cranium mengadopsi pola *layered architecture*, yang membagi sistem menjadi beberapa lapisan utama: *controller, service*, dan *repository*. Masing-masing lapisan memiliki tanggung jawab tersendiri yang saling terpisah namun terintegrasi.



Gambar 3.2. Hubungan *Controller-Service-Repository* dalam arsitektur ERP Cranium

Seperti ilustrasi pada Gambar 3.2, Lapisan *controller* merupakan titik masuk utama untuk menerima *HTTP request* dari *client*. Tugas utama *controller* adalah mengarahkan data yang diterima ke *service* untuk diproses. Meskipun tidak menjalankan logika bisnis, *controller* dapat menangani validasi data input dasar agar data yang diteruskan telah sesuai dengan struktur permintaan. *Controller* mengkomunikasikan data dalam bentuk DTO.

Service merupakan lapisan yang menjalankan logika bisnis inti. Semua proses pengolahan data terjadi di sini, mulai dari validasi lanjutan, konversi DTO ke *entity*, hingga pengambilan keputusan bisnis sebelum data dikirim ke *repository*. Lapisan ini juga menjadi pusat pemanggilan *mapper* dan bertanggung jawab atas alur data yang kompleks. *Service* harus dirancang

dengan baik karena menyimpan banyak aturan sistem.

Repository merupakan lapisan yang berinteraksi langsung dengan *database*. Dalam ERP Cranium, digunakan JPA *Repository* yang menyediakan fungsi dasar seperti *save*, *findById*, *delete*, dan *findAll*. Selain itu, *repository* juga dapat dikembangkan lebih lanjut menggunakan *custom query* atau anotasi seperti @*Query* untuk kebutuhan *query* yang lebih spesifik. *Repository* memungkinkan *service* mengakses data tanpa perlu menulis *SQL* secara eksplisit, sehingga menjaga efisiensi dan kejelasan kode.

Secara umum, alur data dimulai dari permintaan *client* yang dikirim ke *controller*, diteruskan ke *service* dalam bentuk DTO, lalu diubah menjadi *entity* yang akan diproses atau disimpan melalui *repository*. Data yang akan dikembalikan ke *client* juga mengikuti jalur sebaliknya, memastikan struktur yang rapi dan modular.

3. *Unit Test & Contract Test* Untuk memastikan stabilitas dan kualitas perangkat lunak, pengembangan ERP Cranium melibatkan proses pengujian otomatis melalui *unit test* dan *contract test*. *Unit test* bertujuan menguji bagian terkecil dari sistem, seperti satu fungsi atau metode pada class tertentu, secara terpisah. Pengujian ini penting untuk memastikan bahwa komponen *internal* sistem berjalan sesuai ekspektasi tanpa bergantung pada komponen lain. Di sisi *back-end*, *unit test* dijalankan menggunakan *framework* seperti *JUnit*, *Mockito*, dan *Spring Boot Test*.

   Selain *unit test*, digunakan pula *contract test* untuk memastikan kesesuaian komunikasi antara bagian *front-end* dengan *back-end*, terutama pada *controller*. *Contract test* memastikan bahwa *endpoint API* mengembalikan respons dengan struktur dan format sesuai kontrak yang telah disepakati antara pengembang *front-end* dan *back-end*. *Library* seperti *Spring Cloud Contract*, *MockMVC*, dan *JUnit* digunakan untuk memverifikasi bahwa controller memenuhi kontrak tersebut dan menangani permintaan dengan benar.

   Di sisi *front-end*, pengujian dilakukan untuk memverifikasi bahwa komponen *UI* mampu di-*render* dengan baik, merespons input pengguna, dan menampilkan data secara akurat. *Library* seperti *Jest* dan *React Testing Library* digunakan untuk melakukan unit test pada komponen-komponen *React* di *Next.js*. Dengan kombinasi pengujian yang menyeluruh dari sisi

back-end dan *front-end*, pengembangan ERP Cranium menjadi lebih terjamin dari sisi reliabilitas, skalabilitas, dan keamanan sistem.

3.4.4 Pengembangan Modul Master

Modul *master* dalam sistem ERP Cranium merupakan komponen fundamental yang menyimpan data referensi utama yang digunakan oleh modul-modul lain dalam sistem. Modul ini berfungsi sebagai sumber data inti yang menjadi dasar bagi seluruh proses operasional dan transaksi dalam sistem ERP. Karena sifatnya yang sentral dan terhubung langsung dengan banyak modul lain, ketepatan, kelengkapan, dan fleksibilitas modul *master* sangat menentukan kestabilan dan efisiensi alur kerja ERP secara keseluruhan. Oleh karena itu, pengembangan fitur dalam modul *master* menjadi hal yang penting untuk memastikan sistem dapat beradaptasi dengan kebutuhan bisnis yang dinamis, termasuk penyempurnaan input field dan validasi agar data yang tersimpan tetap akurat, konsisten, dan sesuai standar bisnis klien.

A Analisis Masalah

Mengingat banyaknya data bersifat identifikasi yang akan diinput dan diolah dalam praktik penggunaan ERP, sistem ERP berpotensi mengalami duplikasi data. Hal ini dapat menyebabkan inkonsistensi informasi, kesalahan dalam proses transaksi, dan kebingungan dalam pelaporan. Misalnya, jika dua kendaraan memiliki nomor identifikasi yang sama, proses pelacakan aset atau penjadwalan operasional dapat terganggu. Hal yang sama berlaku pada *Warehouse*, *supplier*, *sales*, dan *department* yang menjadi elemen penting dalam alur logistik dan manajemen *internal*. Duplikasi nama atau identitas pada entitas tersebut dapat menciptakan konflik pada saat integrasi antar modul serta menurunkan kepercayaan pengguna terhadap keakuratan sistem.

B Solusi

Untuk mengatasi potensi duplikasi data dalam sistem ERP, solusi yang diimplementasikan adalah penambahan validasi pada sisi *back-end* untuk memastikan bahwa nilai-nilai yang bersifat unik. Seperti nomor kendaraan, nama *warehouse*, nama *supplier*, nama *sales*, dan nama *department* agar tidak dapat dimasukkan lebih dari satu kali. Dengan adanya validasi, sistem dapat menghindari

duplikasi data yang dapat menyebabkan konflik pada transaksi, laporan keuangan yang tidak akurat, atau ketidaksesuaian antar modul. Validasi juga memberikan jaminan bahwa data yang disimpan benar-benar merepresentasikan entitas yang unik dan sah, seperti satu kendaraan hanya boleh memiliki satu nomor registrasi atau satu gudang hanya memiliki satu nama. Selain itu, validasi meningkatkan efisiensi kerja karena pengguna dapat langsung mengetahui kesalahan input tanpa perlu menunggu proses *back-end*, sehingga mempercepat proses entri data dan mengurangi beban koreksi di kemudian hari.

```
@NoArgsConstructor(access = AccessLevel.PRIVATE)
public class MasterUtil {

    public static String normalizeString(String name) {
        if (name == null) {
            return null;
        }
        return name.trim().replaceAll("\\s+", " ");
    }
}
```

Gambar 3.3. Potongan kode *masterUtil*

Yang pertama dilakukan dalam proses penambahan validasi adalah pembuatan utilitas normalisasi *string* dalam *masterUtil*, seperti yang ditunjukkan pada Gambar 3.3. Fungsi bantu ini bertujuan untuk mencegah terjadinya duplikasi data akibat perbedaan format penulisan. Utilitas ini akan menghapus spasi berlebihan dan *whitespace* yang tidak diperlukan pada *input* teks, sehingga nilai yang dimasukkan ke dalam sistem akan memiliki format yang seragam. Langkah ini penting untuk menjaga konsistensi data dan memastikan bahwa validasi keunikan dapat bekerja secara optimal.

```java
@Component
@NoArgsConstructor(access = AccessLevel.PRIVATE)
public class VehicleServiceUtil {

    private static VehicleService vehicleService;

    @Autowired
    public VehicleServiceUtil(VehicleService vehicleService) {
        VehicleServiceUtil.setVehicleService(vehicleService);
    }

    private static void setVehicleService(VehicleService vehicleService) {
        VehicleServiceUtil.vehicleService = vehicleService;
    }

    public static VehicleService getVehicleService() {
        return vehicleService;
    }
```

Gambar 3.4. Potongan kode *MasterVehicleNoIsUnique*

Tahap selanjutnya adalah membuat *VehicleServiceUtil*. *VehicleServiceUtil* adalah kelas utilitas yang memungkinkan akses statis terhadap bean *VehicleService*, terutama untuk digunakan di kelas-kelas yang tidak dikelola langsung oleh Spring seperti *validator*. Karena *validator* tidak bisa menggunakan @*Autowired* secara langsung, maka *VehicleService* di*inject* melalui konstruktor @*Autowired* ke dalam variabel statik di kelas ini. Dengan begitu, komponen *non-Spring* tetap dapat mengakses *VehicleService* melalui method *getVehicleService()*, sehingga proses validasi tetap dapat berjalan dengan baik. Implementasi dari pendekatan ini dapat dilihat pada Gambar 3.4.

Setelah penerapan menambahkan *VehicleServiceUtil*, sistem ERP Cranium juga menerapkan validasi tambahan untuk menjamin keunikan data pada entitas tertentu, salah satunya pada entitas *Vehicle*. Untuk tujuan tersebut, anotasi khusus bernama @*MasterVehicleNoIsUnique* digunakan. Anotasi ini menandakan bahwa nilai dari nomor kendaraan yang dikirimkan saat proses pembuatan maupun pembaruan harus bersifat unik dan tidak boleh sudah ada dalam sistem. Anotasi ini tidak bekerja sendiri, tetapi dikaitkan dengan sebuah *validator* bernama *MasterVehicleNoIsUniqueValidator* yang bertugas memproses dan mengevaluasi keunikan nomor kendaraan berdasarkan data yang ada.

```java
@Documented
@Constraint(validatedBy = {})
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface MasterVehicleNoIsUnique {
    String message() default "{master.vehicle.vehicleno.unique}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Gambar 3.5. Potongan kode *VehicleServiceUtil*

Anotasi @*MasterVehicleNoIsUnique* pada Gambar 3.5 berfungsi sebagai penanda bahwa validasi khusus diperlukan untuk menjamin keunikan nomor kendaraan (*vehicleNo*) dalam proses pembuatan atau pembaruan data kendaraan pada sistem ERP. Anotasi ini merupakan bagian dari mekanisme validasi berbasis *Java Bean Validation* (*Jakarta Validation*), yang ketika digunakan akan memicu *validator* khusus untuk memeriksa apakah nomor kendaraan yang dimasukkan sudah pernah tercatat dalam sistem. Dengan demikian, anotasi ini membantu mencegah duplikasi nomor kendaraan yang dapat menyebabkan ketidakkonsistenan data.

Selanjutnya, pengembangan dilanjut dengan penambahan kelas *MasterVehicleNoIsUniqueValidator* yang merupakan implementasi dari mekanisme validasi kustom pada *Java Bean Validation* yang digunakan untuk memastikan bahwa nomor kendaraan (*vehicleNo*) bersifat unik pada saat proses pembuatan (*create*) atau pembaruan (*update*) data kendaraan dalam sistem ERP Cranium. *Validator* ini dihubungkan langsung dengan anotasi @*MasterVehicleNoIsUnique*, yang ditambahkan pada *controller* atau *service* yang membutuhkan validasi keunikan nomor kendaraan. Dengan memanfaatkan *VehicleService*, kelas ini melakukan pemeriksaan ke basis data apakah nomor kendaraan yang dimasukkan sudah pernah tercatat. Jika nomor kendaraan sudah ada dan bukan milik entitas yang sedang diperbarui, maka validasi akan gagal, yang mencegah duplikasi data yang dapat menimbulkan konflik pada sistem logistik dan operasional. Berikut adalah penjelasan tentang mekanisme dari *MasterVehicleNoIsUniqueValidator* yang mencakup validasi saat proses *create* dan juga *update*:

```
    private boolean updateValidator(Object[] value) {
        Optional<VehicleUpdateDto> vehicleUpdateDtoOptional =
Optional.ofNullable((VehicleUpdateDto) value[0]);
        Optional<Long> vehicleIdOptional = value.length > 1 && value[1] instanceof Long
                ? Optional.ofNullable((Long) value[1])
                : Optional.empty();
        if (vehicleUpdateDtoOptional.isEmpty() || vehicleIdOptional.isEmpty()) {
            return false;
        }

        VehicleUpdateDto vehicleUpdateDto = vehicleUpdateDtoOptional.get();
        Long vehicleId = vehicleIdOptional.get();

        try {
            VehicleDto vehicleDto =
vehicleService.findByVehicleNo(vehicleUpdateDto.getVehicleNo());
            return vehicleDto.getId().equals(vehicleId);
        } catch (DataNotFoundException e) {
            return true;
        }
    }
```

Gambar 3.6. *Method updateValidator* pada *MasterVehicleNoIsUniqueValidator*

UpdateValidator bertugas memvalidasi data kendaraan pada proses pembaruan. Metode ini memeriksa apakah *vehicleNo* yang dimasukkan sudah pernah tercatat di sistem dan memastikan bahwa jika ada, data tersebut memang milik kendaraan yang sedang diperbarui. Hal ini dilakukan dengan mengambil *VehicleUpdateDto* dan *vehicleId* dari *parameter*, lalu memanggil *vehicleService.findByVehicleNo(...)*. Jika data ditemukan, maka akan dibandingkan ID-nya dengan ID kendaraan yang sedang diproses. Validasi hanya lolos jika keduanya cocok. Jika data tidak ditemukan (melempar *DataNotFoundException*), maka dianggap valid karena *vehicleNo* belum digunakan kendaraan lain. Implementasi dari metode ini ditampilkan pada Gambar 3.6.

```
    private boolean createValidator(Object[] value) {
        Optional<VehicleCreateDto> vehicleCreateDtoOptional =
Optional.ofNullable((VehicleCreateDto) value[0]);

        if (vehicleCreateDtoOptional.isEmpty()) {
            return false;
        }

        try {
            VehicleCreateDto vehicleCreateDto = vehicleCreateDtoOptional.get();
            vehicleService.findByVehicleNo(vehicleCreateDto.getVehicleNo());
            return false;
        } catch (DataNotFoundException e) {
            return true;
        }
    }
```

Gambar 3.7. *Method createValidator* pada *MasterVehicleNoIsUniqueValidator*

Sementara itu, *createValidator* digunakan untuk validasi nomor kendaraan saat proses pembuatan data baru. Seperti yang ditampilkan pada Gambar 3.7, fungsi ini pertama-tama memeriksa apakah objek *VehicleCreateDto* tersedia. Jika tersedia, sistem akan mencoba mencari data kendaraan dengan nomor yang sama menggunakan *vehicleService.findByVehicleNo(...)*. Jika data ditemukan, berarti nomor kendaraan telah digunakan dan validasi gagal. Namun jika data tidak ditemukan (ditangkap sebagai *DataNotFoundException*), maka validasi dianggap berhasil karena nomor kendaraan tersebut belum digunakan sebelumnya. Dengan mekanisme ini, sistem memastikan bahwa setiap nomor kendaraan yang baru dimasukkan benar-benar unik.

```
        @Transactional(value = "masterTransactionManager", readOnly = true)
        public VehicleDto findByVehicleNo(String vehicleNo) throws DataNotFoundException {
                String normalized = MasterUtil.normalizeString(vehicleNo);
                Optional<Vehicle> vehicleOptional =
vehicleRepository.findByVehicleNoIgnoreCaseAndDeletedFalse(normalized);
                if (vehicleOptional.isEmpty()) {
                        throw new
DataNotFoundException(masterMessageSource.getMessage(MASTER_VEHICLE_FINDVEHICLENO_NOTFOUND,
new Object[]{normalized}, LocaleContextHolder.getLocale()));
                }
                return vehicleMapper.map(vehicleOptional.get(), VehicleDto.class);
        }
```

Gambar 3.8. *Method findByVehicleNo* pada *VehicleService*

Selanjutnya, pengembangan dilanjutkan dengan penambahan *method findByVehicleNo* yang krusial untuk validasi, sebagaimana ditunjukkan pada Gambar 3.8. *Method findByVehicleNo* pada kelas service digunakan untuk mencari entitas Vehicle berdasarkan *vehicleNo* yang telah dinormalisasi menggunakan *MasterUtil.normalizeString()*. Metode ini bersifat *read-only* dan dilakukan melalui *vehicleRepository.findByVehicleNoIgnoreCaseAndDeletedFalse()*, yang artinya pencarian hanya akan mengembalikan data kendaraan yang belum dihapus (*deleted = false*) dan dengan nomor kendaraan yang sesuai, tanpa memperhatikan huruf kapital. Jika kendaraan tidak ditemukan, maka akan dilemparkan *DataNotFoundException*, sementara jika ditemukan, data tersebut akan di-mapping ke objek *VehicleDto* dan dikembalikan.

Dalam konteks *validator* seperti *MasterVehicleNoIsUniqueValidator*, *method findByVehicleNo* berfungsi sebagai mekanisme untuk mengecek apakah nomor kendaraan yang dimasukkan pengguna sudah ada sebelumnya di *database*. Pada proses validasi data baru (*createValidator*), *method* ini akan dipanggil untuk memeriksa keberadaan nomor kendaraan tersebut—jika ditemukan, validasi gagal karena data tidak unik. Sebaliknya, pada proses *update* (*updateValidator*), method ini memastikan bahwa nomor kendaraan yang akan diperbarui tidak dimiliki oleh entitas lain selain dirinya sendiri, sehingga tetap menjaga keunikan data kendaraan dalam sistem ERP.

Langkah terakhir yang dilakukan dalam proses validasi adalah mengimplementasikan anotasi *validator*, seperti @*MasterVehicleNoIsUnique*, ke dalam *service* dan *controller* yang terkait. Implementasi ini dilakukan dengan menambahkan anotasi validasi tersebut pada parameter *method* di *controller* atau service yang menerima *VehicleCreateDto* atau *VehicleUpdateDto*, baik secara langsung maupun dalam bentuk *array* parameter. Dengan demikian, setiap kali endpoint untuk membuat atau memperbarui data kendaraan diakses, sistem akan secara otomatis memeriksa validitas data menggunakan *validator* yang telah dikembangkan sebelumnya. Jika terdapat pelanggaran seperti duplikasi nomor kendaraan, sistem akan segera memberikan pesan kesalahan sebelum data diteruskan ke tahap penyimpanan. Langkah ini memastikan integritas data tetap terjaga dan mencegah entri data yang tidak *valid* masuk ke dalam sistem.

Seluruh mekanisme *validator* yang telah dijelaskan sebelumnya, seperti pada validasi *MasterVehicleNoIsUnique*, juga akan diaplikasikan ke entitas lain yang memiliki karakteristik data unik, yaitu nama gudang (*warehouse*), nama *supplier*, nama *sales*, dan nama *department*. Prinsip dan struktur logikanya tetap

sama, yakni menggunakan anotasi kustom, *class validator*, *service util*, serta pemanggilan *service* untuk mengecek keberadaan data sebelum menyimpannya. Namun, untuk masing-masing entitas akan dilakukan penyesuaian nama *class, DTO, service*, serta *field* yang divalidasi. Misalnya, *VehicleService* akan digantikan dengan *WarehouseService*, *SupplierService*, *SalesService*, atau *DepartmentService* sesuai konteksnya, begitu pula DTO-nya. Dengan penyesuaian ini, sistem ERP Cranium dapat menjamin bahwa semua entitas penting dengan data identifikasi tunggal terlindungi dari duplikasi, sehingga menjaga konsistensi dan integritas data dalam skala sistem yang lebih luas.

3.4.5 Pengembangan Modul Selling

Modul *Selling* dalam sistem ERP Cranium merupakan komponen yang dirancang untuk menangani seluruh proses penjualan pada sistem. Modul ini memainkan peran penting dalam siklus pendapatan perusahaan karena berkaitan langsung dengan transaksi keluar dan arus kas. Seiring meningkatnya jumlah transaksi dan kebutuhan perusahaan akan efisiensi operasional, pengembangan pada modul ini menjadi krusial untuk meningkatkan kecepatan, akurasi, dan konsistensi data dalam proses penjualan. Fokus pengembangan pada modul *Selling* diarahkan pada pencegahan duplikasi dokumen retur penjualan. Pengembangan ini memperkuat integritas data transaksi serta mencegah terjadinya inkonsistensi yang dapat berdampak pada laporan keuangan maupun stok barang.

A Analisis Masalah

Dalam sistem ERP, proses pembuatan dokumen retur penjualan *Selling Order Return* harus dilakukan dengan ketelitian tinggi untuk memastikan setiap retur benar-benar merujuk pada transaksi penjualan yang sah dan belum pernah diproses sebelumnya. Tanpa mekanisme pencegahan yang jelas, pengguna berisiko membuat dokumen retur ganda untuk satu dokumen *Selling Order* yang sama. Hal ini dapat menyebabkan data transaksi menjadi tidak akurat, mengganggu pencatatan stok barang, serta memunculkan inkonsistensi dalam laporan penjualan dan keuangan. Untuk mengatasi potensi duplikasi tersebut, diperlukan pengembangan fitur yang mampu secara otomatis menyaring dokumen *Selling Order* yang telah digunakan dalam proses retur, sehingga hanya dokumen yang *valid* dan belum diproses yang dapat dipilih oleh pengguna.

B Solusi

Untuk mencegah duplikasi dokumen retur penjualan, sistem memanfaatkan *field* yang telah tersedia pada struktur basis data tabel *selling order*, yaitu *order_do_fulfilled*. *Field* ini berfungsi sebagai penanda status pemrosesan dokumen *Selling Order* terkait. Ketika pengguna membuat dokumen *Selling Order Return*, sistem akan mengubah nilai *order_do_fulfilled* pada *Selling Order* yang dipilih menjadi *true*, menandakan bahwa dokumen *Selling Order* tersebut telah digunakan dalam proses retur dan tidak dapat dipilih kembali. Dengan mekanisme ini, proses pengambilan data *Selling Order* saat pembuatan *Selling Order Return* difilter berdasarkan kondisi *order_do_fulfilled* = *false*, sehingga hanya *Selling Order* yang belum pernah digunakan sebagai referensi retur yang tersedia dalam daftar pilihan.

Selain itu, sistem juga mengakomodasi perubahan status dokumen retur. Apabila dokumen *Selling Order Return* diubah statusnya menjadi *CANCELLED* atau *VOID*, maka nilai *order_do_fulfilled* dari *Selling Order* yang bersangkutan akan dikembalikan menjadi false. Dengan demikian, dokumen *Selling Order* tersebut kembali tersedia untuk diproses dalam retur yang baru. Pendekatan ini tidak hanya menjamin konsistensi data, tetapi juga menjaga fleksibilitas dalam pengelolaan transaksi retur penjualan secara aman dan terkendali.

Tahap awal pengembangan pencegahan duplikasi dokumen *Sales Order Return* dimulai dengan merubah metode pengambilan data *Selling Order* saat create *Selling Order Return* melalui *Repository*. Implementasi perubahan tersebut dapat dilihat pada Gambar 3.9.

```
-     String findOrderDocumentNoQuery = "SELECT id AS id, document_no AS documentNo " +
-             "FROM \"selling\".\"orders\" WHERE deleted = false AND status = 20";
-     @Query(value = findOrderDocumentNoQuery, nativeQuery = true)
-     Optional<List<OrdersDocumentNo>> findOrderDocumentNo();
+
+     Optional<Orders> findByIdAndDeletedFalseAndSellingOrderDeliveryOrderFulfilledIsFalse(@NotNull(message="
  {selling.sellingordersid.notnull}") @Positive(message ="{selling.sellingordersid.positive}") Long id);
+
+     String findUnfulfilledDeliveryOrderQuery = "SELECT id AS id, document_no AS documentNo " +
+             "FROM \"selling\".\"orders\" WHERE deleted = false AND order_do_fulfilled = false";
+     @Query(value = findUnfulfilledDeliveryOrderQuery, nativeQuery = true)
+     Optional<List<OrdersDocumentNo>> findOrdersReturnDocumentNoWithDeliveryOrderNotFulfilled();
```

Gambar 3.9. Perubahan pada *VehicleRepository*

Dari yang mulanya mencari data yang belum dihapus (*findByIdAndDeletedFalse*) menjadi mencari data yang belum dihapus dan memiliki *SellingOrderDeliveryOrderFulfilled* yang bernilai *False*. Perlu diperhatikan bahwa

SellingOrderDeliveryOrder representasi dari *order_do_fulfilled* pada *database Order*. Dengan demikian, sistem mampu memastikan bahwa setiap dokumen retur penjualan yang dibuat berasal dari dokumen penjualan yang valid, yakni yang belum diproses ataupun yang dokumen retur sebelumnya telah dibatalkan.

Setelah itu, pengembangan dilanjut dengan melakukan penyesuaian *OrdersReturnService* agar perubahan bisa diimplementasikan ke dalam sistem. Perubahan ini mencakup modifikasi metode pemanggilan *Selling Order*, seperti yang sudah dirubah pada *repository*. Dilanjut dengan menambah logika yang menyatakan *SellingOrderDeliveryOrder* menjadi *true* secara otomatis ketika dokumen *Sales Order* Return baru berhasil dibuat. Langkah ini dilakukan guna menjaga konsistensi status dokumen dan mencegah terjadinya duplikasi proses retur terhadap dokumen yang sama.

3.4.6 Pengembangan Modul Warehouse

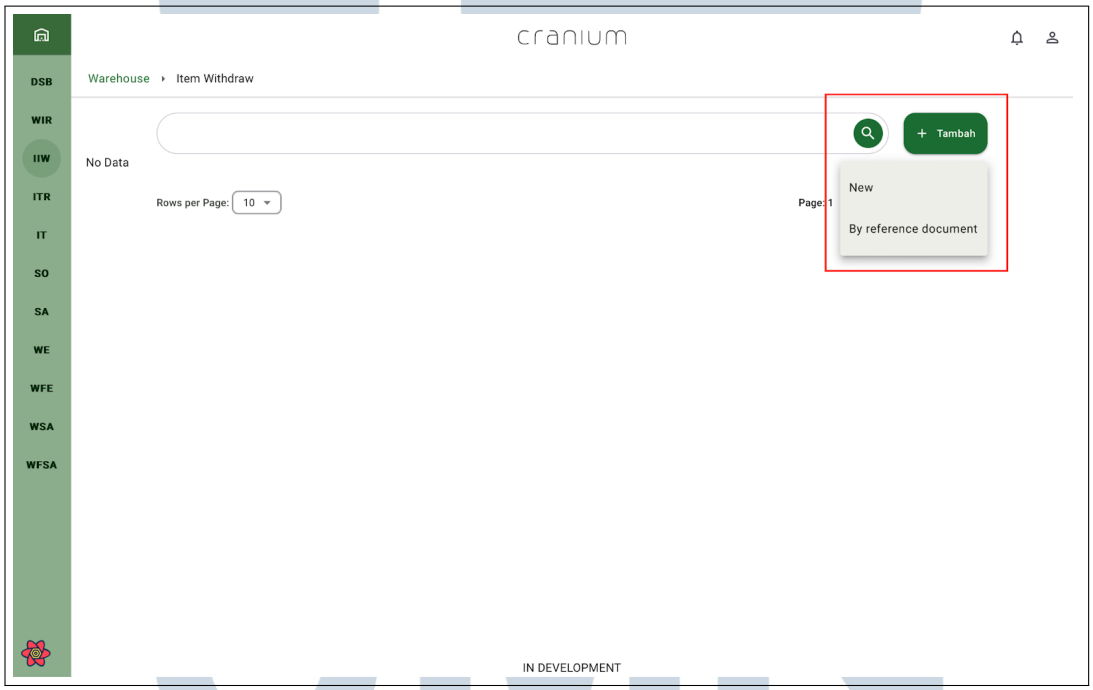
Modul *Warehouse* dalam sistem ERP Cranium berperan penting dalam pengelolaan data gudang yang berkaitan dengan aktivitas logistik dan inventaris perusahaan. Keakuratan data pada modul ini sangat krusial karena menjadi dasar bagi proses bisnis lainnya, seperti transaksi pembelian, penjualan, dan pengiriman. Fokus utama dari pengembangan pada modul *Warehouse* mencakup penghapusan fitur serta perbaikan terhadap beberapa *bugs* yang mempengaruhi produktifitas pengguna. Penghapusan fitur ini dilakukan sebagai bagian dari penyelarasan alur kerja dengan kebutuhan operasional terkini perusahaan. Selain itu, perbaikan *bugs* ditujukan untuk memastikan bahwa proses pencatatan dan pengelolaan data gudang berjalan lebih andal dan konsisten. Pengembangan ini diharapkan dapat meningkatkan efisiensi modul Warehouse serta memperkuat integrasi antarproses dalam sistem ERP Cranium.

A Penghapusan Fitur

A.1 Analisis Masalah

Penghapusan fitur *Create By Reference Document* pada proses penambahan (*Create*) data di halaman indeks modul *Item Withdraw* dilakukan sebagai respons terhadap permasalahan fungsional dan duplikasi alur yang teridentifikasi dalam implementasi sebelumnya. Fitur tersebut semula memungkinkan pengguna untuk membuat *Item Withdraw* berdasarkan referensi dokumen

ItemWithdrawReservation, namun dalam praktiknya menimbulkan inkonsistensi data dan kompleksitas yang tidak diperlukan dalam alur kerja gudang. Selain itu, entitas *ItemWithdrawReservation* dan *ItemWithdrawReservationDetail* tidak lagi digunakan secara efektif dalam proses bisnis terkini, sehingga kehadirannya dalam sistem justru memperbesar potensi kesalahan pencatatan dan menambah beban pemeliharaan. Oleh karena itu, penghapusan fitur dan entitas terkait menjadi langkah penting untuk menyederhanakan alur kerja, meningkatkan kejelasan antarmuka pengguna, dan memastikan hanya komponen yang relevan dan berfungsi optimal yang dipertahankan dalam sistem ERP Cranium. Implementasi fitur sebelum dilakukan penghapusan dapat dilihat pada Gambar 3.10.



Gambar 3.10. Tampilan modul *warehouse - item withdraw* sebelum dilakukan Penghapusan (munculnya opsi "*by reference document*" saat menekan tombol *create*)

A.2 Solusi

Berdasarkan analisis kebutuhan dan hasil evaluasi terhadap efektivitas modul, penghapusan fitur *Item Withdraw Reservation* dilakukan sebagai langkah penyederhanaan dan peningkatan konsistensi proses kerja pada modul Warehouse. Solusi yang diterapkan meliputi penghapusan opsi *Create by Reference Document* pada saat pengguna melakukan penambahan data (*Add*) di halaman *Index Item Withdraw*.

Di sisi *front-end*, seluruh fitur, halaman, dan komponen yang berkaitan dengan proses *Item Withdraw Reservation* dihapus sepenuhnya. Ini termasuk *folder itemwithdrawreservation* yang berisi *api/header*, *components*, *enums*, *index.ts*, dan *types* pada direktori *features/warehouse*, serta halaman pada *pages/warehouse/item-withdraw-reservation* dan *pages/warehouse/item-withdraw/create-by-item-withdraw-reservation*.

Di sisi *back-end*, proses penghapusan dilakukan secara menyeluruh pada modul *itemwithdrawreservation*, yang mencakup *folder controller*, *entity*, *repository*, *service*, *mapper*, *specification*, dan pengujian di direktori *test/java/id/cranium/erp/warehouse*. Semua dependensi dan referensi terhadap entitas *ItemWithdrawReservation* dan *ItemWithdrawReservationDetail* juga dibersihkan dari kode.

Dengan tindakan ini, sistem ERP Cranium menjadi lebih ringkas, meminimalkan beban pemeliharaan, dan mencegah potensi inkonsistensi data akibat duplikasi alur proses reservasi yang tidak lagi digunakan.

B Perbaikan Bugs

B.1 Analisa Masalah

Permasalahan yang ditemukan pada modul *Warehouse* berkaitan dengan hilangnya jejak audit atau *audit trail* pada beberapa submodul penting, seperti *Item Withdraw*, *Item Transfer Reservation*, *Item Receipt*, dan *Stock Opname*. *Audit trail* merupakan komponen krusial dalam sistem ERP karena berfungsi mencatat aktivitas pengguna dan perubahan status data dalam sistem secara historis. Ketiadaan data audit ini dapat menyebabkan berkurangnya transparansi, menyulitkan proses pelacakan jika terjadi kesalahan atau ketidaksesuaian data, serta menurunkan keandalan sistem dalam konteks *audit* internal maupun eksternal. Oleh karena itu, keberadaan *audit trail* yang lengkap dan konsisten sangat diperlukan guna mendukung integritas serta akuntabilitas sistem dalam pengelolaan gudang.

B.2 Solusi

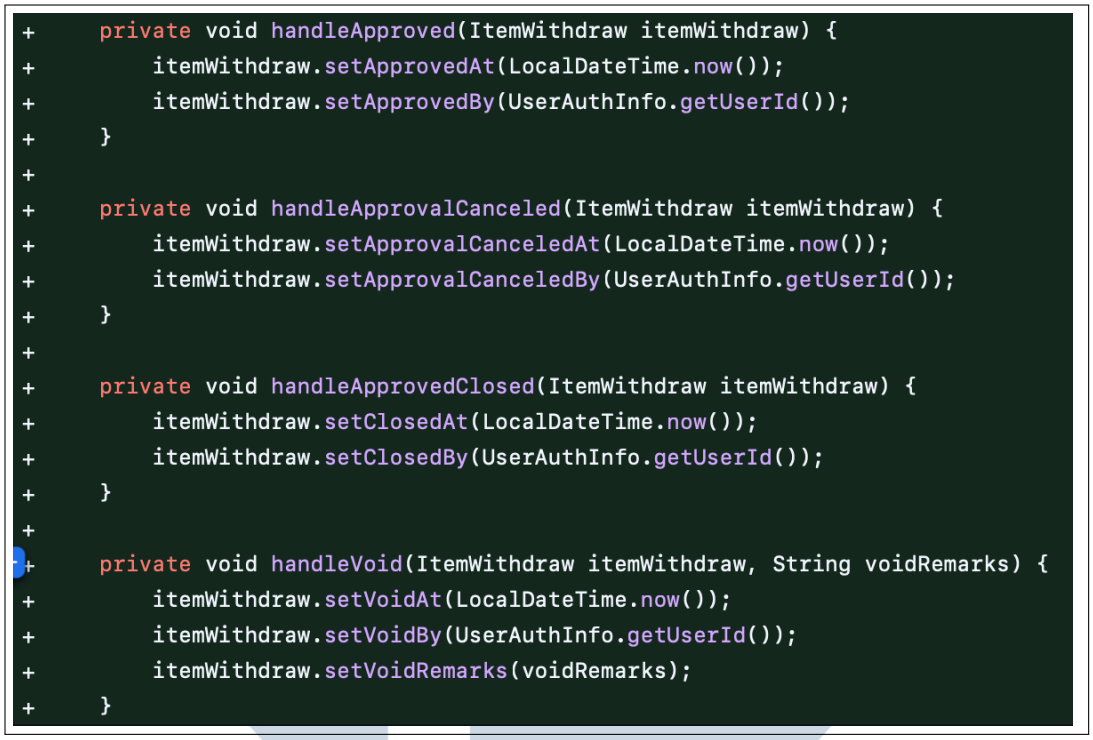
Sebagai solusi atas hilangnya *audit trail* di beberapa submodul pada modul *Warehouse*, dilakukan penelusuran menyeluruh terhadap implementasi audit di masing-masing submodul, seperti *Item Withdraw*, *Item Transfer Reservation*, *Item Receipt*, dan *Stock Opname*. Perbaikan dimulai dengan memastikan bahwa

entitas-entitas terkait telah mewarisi struktur audit secara benar, seperti *field approvedBy*, *voidBy*, dan *voidremarks*. Selanjutnya, dilakukan pengecekan dan penyesuaian pada proses penyimpanan dan pembaruan data di lapisan *service* agar *audit field* dapat terisi secara otomatis saat terjadi perubahan. Di sisi *front-end*, dilakukan penyesuaian tampilan untuk menampilkan kembali informasi audit yang sebelumnya hilang. Dengan langkah-langkah ini, sistem kembali mampu mencatat dan menampilkan riwayat aktivitas secara lengkap, sehingga menjamin transparansi dan integritas data yang lebih baik. Berikut adalah uraian dari perbaikan *bugs* pada *audit trail* di setiap submodul *Warehouse*:

- Submodul *Item Withdraw*
  Pada submodul *Item Withdraw*, ditemukan permasalahan di mana sejumlah status *audit trail* tidak dapat ditampilkan secara semestinya di sisi klien. Permasalahan ini disebabkan oleh belum lengkapnya implementasi kode yang berfungsi untuk menangani perubahan status secara menyeluruh dalam alur kerja sistem. Setelah dilakukan analisis, diketahui bahwa belum terdapat *method handler* yang secara khusus mengatur penyimpanan status beserta informasi tambahan yang berkaitan, seperti waktu perubahan status dan pengguna yang melakukan perubahan tersebut. Oleh karena itu, solusi yang diambil adalah menambahkan *method* khusus dalam *ItemWithdrawService* yang berfungsi untuk menangani setiap perubahan status dan memastikan data tersebut disimpan dengan lengkap ke dalam sistem. Implementasi penambahan pemanggilan *handler* serta detail penanganan perubahan status untuk masing-masing kondisi dapat dilihat pada Gambar 3.11 dan Gambar 3.12.

```
+         Status status = Status.getEnum(itemWithdrawUpdateStatusDto.getStatus());
+
+         if (status != null) {
+             switch (status) {
+                 case APPROVED -> handleApproved(itemWithdraw);
+                 case APPROVED_CLOSED -> handleApprovedClosed(itemWithdraw);
+                 case APPROVED_CANCELED -> handleApprovalCanceled(itemWithdraw);
+                 case VOID -> handleVoid(itemWithdraw,
  itemWithdrawUpdateStatusDto.getVoidRemarks());
+             }
+         }
```

Gambar 3.11. Penambahan pemanggilan *handler* saat *update status*



```
+     private void handleApproved(ItemWithdraw itemWithdraw) {
+         itemWithdraw.setApprovedAt(LocalDateTime.now());
+         itemWithdraw.setApprovedBy(UserAuthInfo.getUserId());
+     }
+
+     private void handleApprovalCanceled(ItemWithdraw itemWithdraw) {
+         itemWithdraw.setApprovalCanceledAt(LocalDateTime.now());
+         itemWithdraw.setApprovalCanceledBy(UserAuthInfo.getUserId());
+     }
+
+     private void handleApprovedClosed(ItemWithdraw itemWithdraw) {
+         itemWithdraw.setClosedAt(LocalDateTime.now());
+         itemWithdraw.setClosedBy(UserAuthInfo.getUserId());
+     }
+
+     private void handleVoid(ItemWithdraw itemWithdraw, String voidRemarks) {
+         itemWithdraw.setVoidAt(LocalDateTime.now());
+         itemWithdraw.setVoidBy(UserAuthInfo.getUserId());
+         itemWithdraw.setVoidRemarks(voidRemarks);
+     }
```

Gambar 3.12. Penambahan *handler* untuk masing masing perubahan *status*

Method handler diperlukan untuk menangani setiap perubahan status yang terjadi dalam alur proses *Item Withdraw*. Fungsinya adalah untuk mencatat status baru beserta detail penting lainnya seperti waktu terjadinya perubahan (*timestamp*) dan identitas pengguna yang melakukan perubahan tersebut.

- Submodul *Stock Opname*

  Pada submodul *Stock Opname*, ditemukan permasalahan di mana informasi *voidRemarks* tidak dapat ditampilkan di sisi klien dan tidak tersimpan ke dalam sistem. Setelah dilakukan proses *debugging* dan penelusuran terhadap alur data, diketahui bahwa akar permasalahan terletak pada ketidakhadiran *field voidRemarks* di dalam *StockOpnameDto*. Karena *field* ini tidak dideklarasikan dalam DTO (*Data Transfer Object*) yang menjadi jembatan antara sistem *back-end* dan *front-end*, maka data *voidRemarks* yang dikirimkan dari antarmuka pengguna tidak dapat diproses lebih lanjut, baik untuk disimpan ke dalam basis data maupun untuk ditampilkan kembali di halaman antarmuka pengguna. Padahal, komponen lain yang berfungsi untuk menangani dan memproses data ini, seperti handler di *controller*, *service method*, hingga konfigurasi *mapper*, sebenarnya telah tersedia dan tidak memiliki kendala. Oleh karena itu, solusi yang dilakukan adalah

menambahkan properti *voidRemarks* ke dalam kelas *StockOpnameDto*, sehingga alur pengiriman dan pengambilan data dapat kembali berjalan normal dan informasi pembatalan dapat tercatat secara transparan dalam sistem.

- Submodul *Item Receipt*

  Pada submodul *Stock Opname*, ditemukan permasalahan terkait tidak munculnya informasi *voidRemarks* pada sisi klien, khususnya saat melakukan proses peninjauan data (*viewing*). Setelah dilakukan penelusuran, diketahui bahwa penyebab utama dari masalah ini bukan terletak pada proses penyimpanan data, karena *field voidRemarks* telah tersedia dan tersimpan dengan baik di sisi *back-end*. Namun, permasalahan terjadi karena belum terdapat *field* atau komponen tampilan yang menampilkan data *voidRemarks* pada antarmuka pengguna, tepatnya di halaman *dashboard-itemstockopname*. Akibatnya, meskipun data *void remarks* telah tercatat dalam sistem, pengguna tidak dapat melihat informasi tersebut pada tampilan aplikasi. Untuk mengatasi hal ini, diperlukan penyesuaian pada sisi *front-end* dengan menambahkan *field* yang menampilkan *voidRemarks* pada halaman yang bersangkutan agar *audit trail* dapat ditampilkan secara utuh dan mendukung kebutuhan *monitoring* proses *stock opname* oleh pengguna.

- Submodul *Item Transfer Reservation*

  Pada submodul *Item Transfer Reservation*, *approvalCanceledAt* dan *approvalCanceledBy* tidak dapat ditampilkan ke sisi klien, hingga tidak dapat tersimpan ke dalam sistem. Meskipun semua method untuk penyimpanan dan penampilan data sudah lengkap, *audit trail* untuk *approvalCanceledAt* dan *approvalCanceledBy* tidak dapat berjalan. Setelah melakukan *debugging*, ditemukan bahwa kedua *field* tersebut belum dimasukkan ke dalam *ItemTransferRequestMapper*. *Mapper* ini berfungsi sebagai penghubung antara entitas dan *Data Transfer Object* (DTO), sehingga jika suatu *field* tidak dimasukkan ke dalam proses pemetaan, maka data tersebut tidak akan terbawa dalam alur input maupun *output* sistem. Akibatnya, meskipun *field* sudah tersedia di entitas dan DTO serta seluruh method penyimpanan dan penampilan data telah dibuat, informasi tersebut tetap tidak dapat diproses oleh sistem secara menyeluruh. Sehingga perubahan yang perlu dilakukan adalah menambah kedua *field* tersebut ke dalam *ItemTransferRequestMapper*. Penyesuaian ini memastikan bahwa data yang berkaitan dengan pembatalan

approval dapat dikirim dan diterima dengan benar antara *back-end* dan *front-end*.

3.5 Kendala dan Solusi

Kendala yang ditemukan selama kerja praktik magang adalah sebagai berikut:

- Kurangnya pendalaman materi mengenai *business process*, sehingga banyak waktu terbuang untuk mempelajari *business process* setiap modul yang dikerjakan.
- *Progress* pekerjaan seringkali terhambat dikarenakan ada *pull request* pekerjaan sebelumnya yang belum di-*merge*.
- Saat mengerjakan *task*, banyak ditemukan *bugs* tak terduga yang harus diperbaiki. Mengakibatkan penyelesaian *task* melewati deadline yang suah ditentukan.

Solusi atas kendala-kendala yang dihadapi selama kerja praktik magang adalah sebagai berikut:

- Melakukan pembelajaran secara mandiri mengenai *business process* yang ada, serta secara proaktif bertanya mengenai *business process* kepada mentor.
- Aktif berdiskusi dengan mentor, *supervisor*, dan tim tentang perkembangan progress untuk memastikan kelancaran workflow dan proses *pull request*.
- Melakukan komunikasi dengan *supervisor* terkait *bugs* yang tak terduga, sehingga dilakukan penyesuaian waktu pengerjaan *task*.