PROGRES
PROYEK AKHIR

Pengembangan Sistem Manajemen Operasional dan Sistem Informasi Layanan Pengguna untuk Kendaraan Listrik Berbasis Arsitektur Modular Monolith

Sultan Argya Safa Firdaus
NRP 3122600022

DOSEN PEMBIMBING

Dr. Ferry Astika Saputra, S.T., M.Sc.
NIP 19770823200112100 2

Dr.Idris Winarno,S.ST, M.Kom
NIP. 198203082008121001

Prof. Ir. Dadet Pramadihanto, M.Eng., Ph.D.
NIP 196202111988111001

PROGRAM STUDI SARJANA TERAPAN TEKNIK INFORMATIKA
DEPARTEMEN TEKNIK INFORMATIKA DAN KOMPUTER
POLITEKNIK ELEKTRONIKA NEGERI SURABAYA
2025

Halaman sengaja dikosongkan

DAFTAR ISI

DAFTAR GAMBAR

Halaman sengaja dikosongkan

BAB 1
PENDAHULUAN

1.1 LATAR BELAKANG

Penggunaan kendaraan listrik atau *Electric Vehicle* (EV) dalam transportasi publik semakin meningkat seiring upaya global menurunkan emisi. Namun, armada EV memiliki tantangan operasional tersendiri, terutama jangkauan baterai yang terbatas sehingga memerlukan penyesuaian jadwal operasional kendaraan dan kru [1]. Untuk mengatasi kendala tersebut, penerapan sistem manajemen armada atau *Fleet Management System* (FMS) menjadi krusial dalam mengoptimalkan pengoperasian bus listrik. Berbagai studi menunjukkan bahwa FMS berbasis teknologi digital dapat meningkatkan efisiensi operasional armada EV, mengoptimalkan penggunaan sumber daya, serta menekan biaya operasional dan waktu henti kendaraan [2]. Dengan kata lain, pemanfaatan FMS mampu mengoptimalkan penjadwalan perjalanan dan perawatan kendaraan, sehingga operasi armada lebih lancar dan produktivitas meningkat [2].

Di Indonesia, salah satu implementasi nyata sistem manajemen armada berbasis digital adalah MitraDarat, sebuah platform FMS yang dikembangkan oleh Kementerian Perhubungan. MitraDarat menyediakan fitur-fitur penting seperti pelacakan kendaraan *real-time*, manajemen jadwal dan rute, serta prediksi waktu kedatangan atau *Estimated Time of Arrival* (ETA) yang dapat diakses oleh operator maupun pengguna [3]. Keberadaan platform seperti MitraDarat menunjukkan pentingnya pengembangan FMS untuk pengelolaan armada transportasi nasional berbasis EV.

Dalam rangka mendukung operasional dan manajemen armada bus listrik secara efektif, Politeknik Elektronika Negeri Surabaya (PENS) bersama mitra industri tengah membangun suatu *Fleet Management System* (FMS) yang terdiri dari beberapa subsistem. Subsistem-subsistem utama tersebut antara lain: *Intelligent Agent System* (IAS) sebagai perangkat pada tiap kendaraan untuk menangkap dan mengirim data sensor secara *real-time*, sistem pemeliharaan untuk memantau data sensor kendaraan dan mencatat riwayat perawatan, sistem operasional untuk mengatur trayek serta mengelola jadwal dan armada bus, dan sistem layanan pelanggan untuk menyajikan informasi kepada penumpang (misalnya posisi bus dan estimasi waktu kedatangan). Masing-masing subsistem memiliki fungsionalitas spesifik sesuai domainnya. Namun, penggunaan banyak subsistem terpisah berpotensi menimbulkan fragmentasi data dan kesulitan integrasi antar komponen. Tantangan integrasi ini dapat menghambat fleksibilitas pengembangan fitur baru pada sistem secara keseluruhan [4].

Oleh karena itu, proyek akhir ini berfokus pada pengembangan sistem manajemen armada kendaraan listrik dengan pendekatan arsitektur *modular monolith*. Berbeda dengan paradigma subsistem terpisah, arsitektur *modular monolith* menggabungkan seluruh fungsionalitas (operasional armada dan layanan pengguna) ke dalam satu kesatuan aplikasi yang utuh, sembari menjaga pemisahan dalam modul-modul yang longgar (*loosely coupled*). Pendekatan ini diharapkan dapat mengatasi masalah integrasi antar fitur, sekaligus meningkatkan fleksibilitas dan kecepatan pengembangan sistem dibandingkan arsitektur monolitik tradisional.

1.2 PERMASALAHAN

Berdasarkan latar belakang yang telah diuraikan, menghasilkan permasalahan yang diidentifikasi sebagai berikut:

1. Sistem operasional armada yang ada saat ini masih belum mampu mendorong efektivitas armada EV. Belum tersedia fitur penjadwalan ulang kendaraan dan kru secara dinamis, manajemen disrupsi operasional, maupun optimalisasi rute untuk memaksimalkan efisiensi operasional dan konsumsi energi. Selain itu, sistem informasi bagi pengguna (*user service*) juga belum optimal, misalnya belum tersedianya estimasi kedatangan bus secara akurat, informasi kapasitas kursi tersedia, dan fitur pendukung layanan pengguna lainnya.
2. Sistem yang sudah ada belum mendukung pembuatan variasi aplikasi FMS yang beragam sesuai kebutuhan. Setiap penyesuaian atau penambahan fungsi pada sistem memerlukan refaktorisasi kode berskala besar, sehingga menghambat kecepatan dan kelincahan proses pengembangan.
3. Belum terdapat kerangka kerja yang mendukung implementasi arsitektur modular dan terintegrasi dengan baik untuk pengembangan fitur baru secara efisien. Akibatnya, tim pengembang mengalami kesulitan dalam mengembangkan atau menambahkan modul fitur baru tanpa mengganggu struktur arsitektur sistem secara keseluruhan.

1.3 TUJUAN

Adapun tujuan dari mengembangkan sistem manajemen armada dengan pendekatan arsitektur *modular monolith* adalah sebagai berikut:

1. Mengembangkan sistem manajemen armada kendaraan listrik yang dapat mengoptimalkan operasional armada EV, mencakup aspek operasional internal operator maupun layanan informasi bagi penumpang.
2. Menerapkan arsitektur *modular monolith* dalam pengembangan sistem, agar sistem yang dibangun mendukung fleksibilitas pengembangan tanpa memerlukan refaktor kode besar-besaran di kemudian hari.
3. Menyediakan struktur sistem dan kerangka kerja yang modular sehingga memungkinkan integrasi fitur-fitur seperti pelacakan armada (fleet tracking), pengelolaan rute dan jadwal operasional, serta estimasi waktu kedatangan (ETA) secara efisien dan berkelanjutan.

1.4 MANFAAT

Manfaat dari proyek akhir ini adalah bagi Operator Armada, yaitu memberikan sistem FMS yang dapat mendukung pengelolaan armada kendaraan listrik secara efektif dan optimal. Bagi Tim Pengembang, proyek ini menyediakan struktur kode dan kerangka kerja yang mendukung pengembangan serta integrasi fitur secara modular sehingga dapat mempercepat proses penambahan fitur baru. Sedangkan bagi Penumpang, proyek ini memberikan akses informasi real-time terkait keberadaan armada, jadwal keberangkatan, serta estimasi waktu tiba kendaraan terhadap halte tujuan secara akurat. Hal ini meningkatkan kenyamanan dan kepercayaan pengguna terhadap layanan transportasi berbasis EV.

1.5 SISTEMATIKA PENULISAN

BAB 1 PENDAHULUAN

Bab ini menguraikan latar belakang masalah, perumusan masalah, tujuan penelitian, manfaat penelitian, serta sistematika penulisan laporan secara keseluruhan.

BAB 2 KAJIAN PUSTAKA

Bab ini berisi tinjauan teori-teori yang relevan dan hasil penelitian terdahulu yang mendukung pengembangan sistem manajemen armada kendaraan listrik serta pendekatan arsitektur modular monolith. Pembahasan mencakup konsep-konsep FMS pada kendaraan listrik, manajemen operasional armada, serta prinsip-prinsip arsitektur modulith dibandingkan arsitektur monolith dan microservices dari literatur yang kredibel.

BAB 3 DESAIN SISTEM

Bab ini menjelaskan metode yang digunakan dalam pelaksanaan penelitian dan pengembangan sistem. Di dalamnya termasuk metodologi perancangan dan pengembangan perangkat lunak, perangkat atau tool yang digunakan, serta tahapan implementasi proyek akhir secara terstruktur..

BAB 4 EKSPERIMEN DAN ANALISIS

Bab ini memaparkan hasil implementasi sistem yang telah dikembangkan beserta skenario uji coba yang dilakukan. Hasil pengujian terhadap fungsi-fungsi utama sistem disajikan untuk mengevaluasi apakah sistem manajemen armada EV yang dibangun telah memenuhi tujuan dan mengatasi permasalahan yang dirumuskan. Bab ini juga mencakup analisis dan pembahasan mengenai efektivitas kerangka kerja modular serta keberhasilan implementasi arsitektur *modular monolith* pada FMS.

BAB 5 PENUTUP

Bab terakhir berisi kesimpulan dari penelitian proyek akhir yang menjawab tujuan yang telah ditetapkan sebelumnya. Selain itu disertakan pula saran-saran untuk pengembangan lebih lanjut, baik untuk penyempurnaan sistem manajemen armada EV ini di masa mendatang maupun rekomendasi penerapan arsitektur modulith pada proyek serupa berikutnya.

Halman sengaja dikosongkan

BAB 2
KAJIAN PUSTAKA

Pada bab ini dijelaskan teori-teori dasar dan penelitian terkait yang menjadi landasan dalam pengembangan FMS. Pembahasan meliputi konsep-konsep teknis, teknologi pendukung, serta penelitian sebelumnya yang relevan.

2.1 DESKRIPSI PERMASALAHAN

Sistem operasional bus listrik yang ada saat ini masih terbatas dan belum mampu mengoptimalkan pengelolaan armada. Banyak fitur penting FMS *(Fleet Management System)* yang belum diimplementasikan, ditambah keberadaan bug dalam sistem, menyebabkan kinerjanya kurang andal ketika menangani skenario di luar alur ideal [5]. Idealnya, FMS yang baik perlu mengintegrasikan manajemen armada dan logistik secara menyeluruh untuk mengurangi kompleksitas operasional [6]. Tanpa fitur-fitur FMS yang memadai misalnya pelacakan kendaraan secara real-time, optimasi rute, dan monitoring kinerja kendaraan operasional armada EV menjadi tidak efisien. Data telematik real-time yang biasanya disediakan oleh FMS memungkinkan penyesuaian cepat (misalnya pengalihan rute untuk menghindari kemacetan) yang dapat menurunkan konsumsi energi dan waktu tempuh [7]. Ketiadaan integrasi fitur semacam itu membatasi fleksibilitas sistem saat ini dan menyulitkan penambahan fitur baru.

Selain kekurangan dari segi fungsional, arsitektur perangkat lunak pada sistem eksisting yang terbagi menjadi beberapa subsistem *(distributed)* turut menjadi hambatan. Setiap penambahan atau modifikasi fitur yang memerlukan integrasi antar subsistem memerlukan perubahan pada banyak bagian, sehingga proses pengembangan menjadi sulit dan mahal. Akan tetapi, arsitektur monolitik tradisional juga membuat perubahan kecil pun dapat berdampak luas, menghambat responsivitas terhadap kebutuhan layanan transportasi yang dinamis. Studi terkini menekankan pentingnya fleksibilitas serta kemampuan integrasi yang tinggi dalam manajemen transportasi modern [6]. Untuk menjawab tantangan tersebut, diperlukan pendekatan arsitektur yang memungkinkan kemudahan penyesuaian dan ekspansi fitur tanpa merombak keseluruhan sistem.

Pendekatan arsitektur *modular monolith* dipilih sebagai solusi, karena mampu menggabungkan keunggulan sistem monolitik tradisional dengan modularitas ala *microservices*. Arsitektur ini mengorganisasi aplikasi sebagai satu kesatuan utuh namun terdiri dari modul-modul terpisah yang longgar terhubung (*loosely coupled*) [8]. Setiap modul memiliki batas tanggung jawab yang jelas dan dependensi yang eksplisit pada modul lain. Seluruh aplikasi tetap di-*deploy* sebagai satu unit, tetapi kode disusun dalam komponen-komponen yang dapat dikembangkan dan diuji secara independen. Pola ini memungkinkan modul besar untuk dipindahkan atau diubah menjadi layanan terpisah di masa depan tanpa perlu merombak arsitektur dari nol. Dengan demikian, *modular monolith* menawarkan fleksibilitas tinggi sekaligus menjaga kohesi sistem yang dapat meningkatkan kelincahan pengembangan FMS. Melalui arsitektur ini, sistem operasional bus listrik diharapkan dapat dikembangkan dengan lebih fleksibel, mudah dikelola, dan siap beradaptasi dengan perubahan kebutuhan di masa mendatang.

2.2 TEORI PENUNJANG

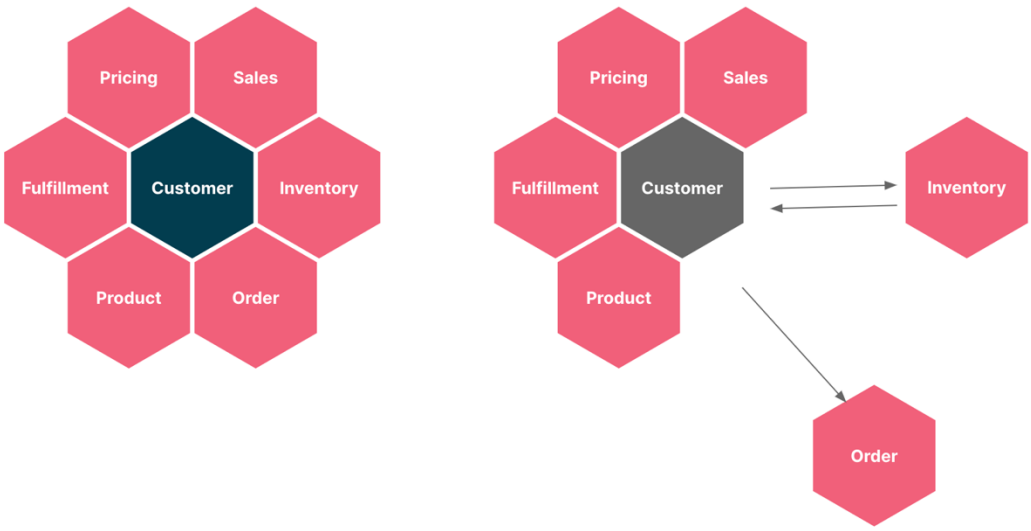
Berikut adalah beberapa teori penunjang yang berhubungan dengan proyek akhir ini:

2.2.1 Fleet Management System

Sistem Manajemen Armada (*Fleet Management System*) adalah sistem terpadu yang mengendalikan operasi armada kendaraan. FMS memantau kendaraan secara real-time, mengelola pemeliharaan, jadwal, dan komunikasi, serta memberikan laporan manajemen. FMS umumnya mencakup pelacakan posisi kendaraan (GPS), optimasi rute, pengelolaan bahan bakar, pengingat perawatan, serta pelaporan performa armada [4]. Hasil penelitian menunjukkan bahwa FMS dapat mengurangi risiko operasional, meningkatkan kualitas layanan, dan memperbaiki efisiensi operasional dengan biaya minimal [4].

2.2.2 Arsitektur Modular Monolith

Arsitektur *modular monolith* adalah pola desain perangkat lunak yang menggabungkan kesederhanaan monolitik dengan modularitas mirip mikroservis. *Modular monolith* menstrukturkan sistem ke dalam modul-modul longgar terhubung, setiap modul memiliki batasan tanggung jawab dan ketergantungan eksplisit pada modul lain [8]. Secara operasional, aplikasi diproduksi dan di-deploy sebagai satu unit tunggal, namun kode diorganisasi ke dalam komponen yang dapat dikembangkan dan diuji secara independen [8]. Dengan pola ini, modul-modul besar dapat dipindah atau diubah menjadi layanan terpisah di masa depan tanpa perlu merombak arsitektur dari nol.



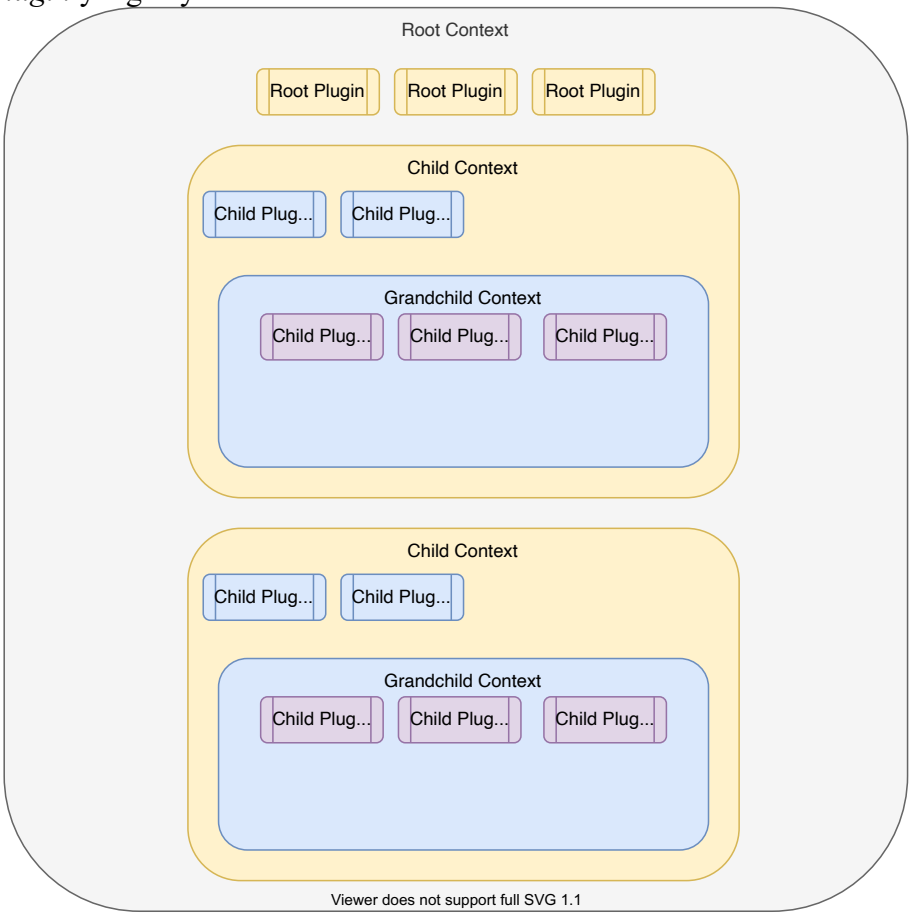
Gambar 2.1 Contoh Penerapan Arsitektur Modular Monolith

Dalam konteks sistem manajemen armada EV, *modular monolith* memungkinkan pembagian fitur kendaraan (misalnya pengelolaan pengisian, telemetri, jadwal, dan antarmuka pengguna) ke dalam modul khusus, namun cukup dijalankan dalam satu aplikasi. Ini menyederhanakan proses pengembangan dan penyebaran awal. Selain itu, *modular monolith* meningkatkan stabilitas dan kinerja karena komponen-komponennya dibuat untuk bekerja lancar satu sama lain [9]. Dengan demikian, arsitektur ini memungkinkan tim riset untuk mengembangkan sistem EV yang kompleks secara bertahap dengan kerangka kerja kode yang

rapi, sekaligus mempermudah migrasi ke *microservices* di masa depan apabila skalabilitas permintaan meningkat [7].

2.2.3 Fastify Backend Framework

Fastify adalah kerangka kerja (*framework*) *backend* pada platform Node.js yang dirancang untuk kinerja tinggi dan pengembangan efisien. Dokumentasi resmi menyatakan bahwa Fastify menitikberatkan pengalaman pengembang optimal dengan *overhead* rendah dan arsitektur *plugin* yang kuat [10]. Fastify mampu melayani hingga 30.000 permintaan per detik pada *hardware* standar berkat model non-blok *(asynchronous)* dan validasi skema JSON bawaan. Fitur-fitur utama Fastify meliputi dukungan pengelolaan *log* terintegrasi, validasi skema, dan ekosistem *plugin* yang kaya.



Gambar 2.2 Fastify Encapsulation

Fastify dipilih karena performanya yang sangat tinggi dan pengelolaan plugin yang memudahkan pengembangan fungsionalitas tambahan. Dibandingkan framework Node.js lain (misalnya Express), Fastify menunjukkan latensi rendah dan *throughput* tinggi [10]. Kemampuan ini penting untuk aplikasi *real-time* yang harus menangani banyak permintaan dalam waktu singkat. Selain itu, arsitektur *plugin* Fastify memfasilitasi modularisasi fitur (misalnya autentikasi, *tracking*) yang dapat diaktifkan secara fleksibel.

Dalam sistem manajemen armada EV, Fastify dapat mengoptimalkan penanganan data telemetri dan permintaan pengguna secara efisien. Kinerja tinggi Fastify memungkinkan *server backend* menampung beban data lokasi kendaraan dan status baterai yang konstan, serta permintaan antarmuka pengguna (misalnya *dashboard monitoring*) tanpa menjadi *bottleneck*.

Penggunaan Fastify juga mendukung arsitektur *modular monolith* karena *plugin system* pada Fastify dapat menambah fungsionalitas khusus tanpa mengganggu inti aplikasi.

2.2.4 Hybrid Storage (PostgreSQL + MongoDB)

Pendekatan *hybrid storage* atau *polyglot persistence* mengacu pada penggunaan lebih dari satu jenis teknologi basis data dalam satu sistem, dengan tujuan memanfaatkan keunggulan masing-masing untuk jenis data yang berbeda [11]. Dalam konteks sistem manajemen armada EV, proyek ini memanfaatkan kombinasi PostgreSQL (basis data relasional) dan MongoDB (basis data NoSQL berorientasi dokumen). Setiap jenis basis data digunakan sesuai karakter datanya. PostgreSQL digunakan untuk data yang terstruktur dan membutuhkan integritas transaksi tinggi (misalnya data pengguna, data armada, jadwal), sedangkan MongoDB digunakan untuk data semi-terstruktur atau tidak terstruktur yang skemanya dapat berubah dan berukuran besar (misalnya *log* sensor kendaraan, histori lokasi, pesan notifikasi).

PostgreSQL adalah RDBMS (*Relational Database Management System*) *open-source* yang telah matang dan diakui andal. PostgreSQL mendukung standar ACID secara penuh serta fitur-fitur canggih SQL (seperti transaksi, *join* kompleks, prosedur tersimpan) [12]. Dengan skema kuat, PostgreSQL memastikan konsistensi dan integritas data sehingga sesuai untuk mengelola data inti operasional yang tidak sering berubah struktur (contoh: skema kendaraan, skema rute, skema pengguna). Sementara itu, MongoDB sebagai basis data dokumen menawarkan fleksibilitas skema dimana data disimpan dalam format JSON (BSON) yang dapat bervariasi antar dokumen, sehingga mudah menyesuaikan ketika ada penambahan jenis data baru [13]. MongoDB juga dirancang untuk skala horizontal tinggi dan performa baik pada beban tulis/baca besar, cocok untuk menampung data telemetri *real-time* dan catatan *log* yang volume-nya terus bertambah

2.2.5 Protokol Komunikasi Client dan Server

Protokol komunikasi antar layanan mengatur cara berbagai komponen sistem saling bertukar data (contoh: klien *mobile* dan *web* dengan *backend*). Beberapa protokol yang digunakan pada sistem manajemen armada antara lain WebSocket, *Server Sent Events* (SSE), dan REST API.

REST (*Representational State Transfer*) adalah gaya arsitektur untuk layanan web berbasis HTTP. Layanan RESTful menyediakan *endpoint* yang merepresentasikan sumber daya, diakses menggunakan metode HTTP standar (GET, POST, dll). REST banyak digunakan karena kesederhanaan dan kemudahannya diimplementasikan. Daya tarik utama REST adalah sifatnya yang *stateless* (tidak menyimpan sesi di server) dan kemampuannya menekankan antarmuka seragam untuk interaksi *client-server* [14].

Server Sent Events (SSE) adalah teknologi push *server-client* satu arah menggunakan koneksi HTTP terbuka. Menurut dokumentasi MDN, dengan SSE "*server* dapat mengirim data baru ke halaman web kapan saja, dengan melakukan *push* pesan ke halaman tersebut". Setiap pesan diterima oleh klien sebagai *event*. SSE berguna untuk notifikasi atau pembaruan *live* di antarmuka pengguna tanpa membutuhkan *polling* konstan [15].

WebSocket adalah protokol komunikasi yang menyediakan saluran komunikasi dua arah (*full-duplex*) secara simultan di atas satu koneksi TCP. WebSocket dimulai dengan *handshake* HTTP, kemudian beralih ke koneksi TCP murni untuk pertukaran pesan tanpa *overhead* protokol HTTP. Protokol ini sangat cocok untuk aplikasi *real-time* karena memungkinkan *server* dan klien saling mengirim pesan kapan saja [16].

Pilihan protokol bergantung pada kebutuhan komunikasi. REST API dipilih untuk antarmuka layanan publik yang mudah diakses oleh klien (*web/mobile*) karena dukungan luas

di internet [14]. SSE dipertimbangkan untuk fitur *push* satu-arah (misalnya notifikasi status kendaraan) karena implementasinya sederhana menggunakan HTTP biasa. WebSocket dipilih untuk kebutuhan komunikasi *real-time* dua arah (misalnya telemetri kendaraan) karena *full-duplex* dan latensinya sangat rendah.

2.2.6 Bun Runtime

Bun adalah JavaScript *runtime* generasi baru yang ditulis menggunakan bahasa Zig dan menggunakan mesin JavaScriptCore (JS *engine* milik WebKit) sebagai eksekutor skrip. Berbeda dengan Node.js yang fokus sebagai *runtime* saja, Bun sejak awal dirancang sebagai all-in-one toolkit yang menggabungkan *runtime*, *bundler*, *test runner*, dan *package manager* dalam satu distribusi. Tujuan utamanya adalah memberikan waktu *start-up* yang jauh lebih cepat, konsumsi memori yang lebih kecil, dan *throughput* tinggi untuk aplikasi web modern.

Dalam konteks FMS, Bun dimanfaatkan sebagai proses utama yang menjalankan *server* Fastify serta mengelola modul bisnis. Karakteristik performa tinggi dan integrasi *toolchain* yang rapat membuat Bun cocok untuk skenario *modular monolith* yang membutuhkan *hot reload*, waktu kompilasi singkat, dan eksekusi TypeScript/JavaScript yang efisien.

2.2.7 Child Process

Node.js menyediakan *Child Process Module* yang memungkinkan aplikasi membuat proses baru di sistem operasi melalui fungsi seperti *spawn*, *exec*, dan *fork*. Setiap child process berjalan dengan event loop dan memori terpisah, tetapi dapat berkomunikasi dengan proses induk menggunakan kanal standar (*stdin/stdout/stderr*) atau kanal pesan khusus seperti IPC. Fitur ini banyak digunakan untuk melakukan offloading tugas berat CPU, menjalankan program eksternal, atau mengisolasi komponen yang membutuhkan dependensi berbeda [17].

Pendekatan *multi-process* Bun–Fastify dan Node–node-rdkafka secara teori sejalan dengan rekomendasi Node.js dimana proses utama menangani HTTP dan orkestrasi modul, sementara *child process* khusus bertugas sebagai Kafka *worker* yang mengonsumsi dan men-*decode* pesan. Dengan cara ini, event loop HTTP tidak terblokir oleh operasi I/O atau deserialization yang berat, sekaligus memungkinkan pemisahan konfigurasi runtime (misalnya Bun untuk server, Node.js untuk pustaka Kafka yang spesifik).

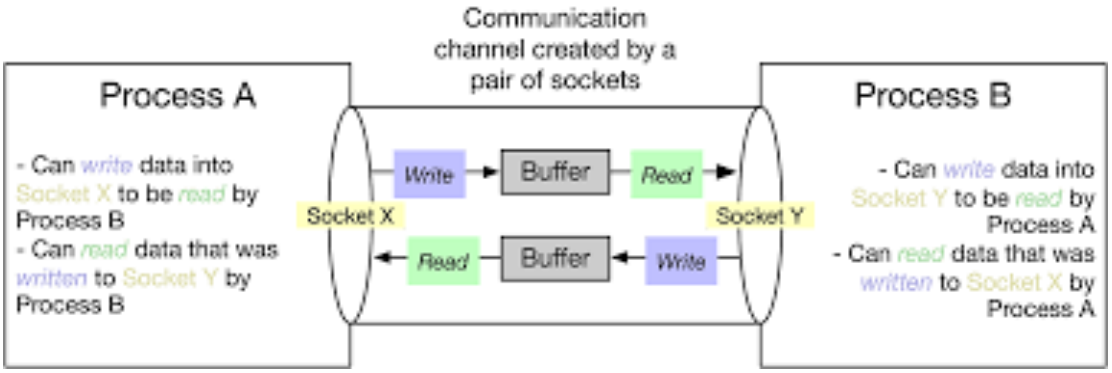
2.2.8 Node-rdkafka

Node-rdkafka adalah binding Node.js untuk *library* C/C++ librdkafka, yang merupakan salah satu klien Apache Kafka berperforma tinggi dan digunakan luas di industri. *Library* ini menyediakan API produser dan konsumer dengan fitur lengkap, termasuk manajemen *offset*, *rebalance consumer group*, kompresi pesan, dan dukungan berbagai strategi *delivery guarantee* [18].

Karena dibangun di atas librdkafka, node-rdkafka mampu memanfaatkan optimasi tingkat rendah (misalnya *zero-copy I/O* dan *batching*) sehingga latensi konsumsi pesan dan penggunaan CPU menjadi lebih efisien dibanding klien murni JavaScript. Hal ini relevan untuk FMS yang harus menangani telematics data dalam jumlah besar secara *near real-time* dari banyak kendaraan.

2.2.9 Unix Domain Socket (UDS) Inter-Process Communication

Unix Domain Socket (UDS) adalah mekanisme *inter-process communication* (IPC) lokal pada sistem operasi Unix. UDS menggunakan *socket address space* khusus (AF_UNIX/AF_LOCAL) dan biasanya diidentifikasi melalui *pathname* pada sistem berkas [19]. Tidak seperti socket TCP/IP yang dirancang untuk komunikasi jaringan, UDS hanya bekerja pada satu *host* sehingga dapat menghindari *overhead* protokol jaringan dan menawarkan *throughput* yang lebih tinggi untuk komunikasi lokal antar proses.



Gambar 2.3 Mekanisme Unix Domain Socket IPC

Secara teoritis, UDS mendukung komunikasi *stream* maupun *datagram*, dan dapat pula membawa *ancillary data* seperti file *descriptor* atau kredensial proses lain. Hal ini menjadikannya pilihan tepat untuk komunikasi antar proses yang saling mempercayai, misalnya antara proses Bun (*runner Fastify*) dan proses Node.js (*Kafka worker*) pada arsitektur FMS. Dengan UDS, pertukaran pesan antar proses dapat dilakukan dengan latensi rendah dan tanpa mengekspos *port* jaringan tambahan, sehingga lebih efisien dan relatif lebih mudah diamankan menggunakan izin file standar POSIX.

2.2.10 Apache Kafka dan Schema Registry

Apache Kafka merupakan platform *distributed event streaming* yang dirancang untuk menangani data *stream* dengan *throughput* tinggi dan latensi rendah secara terdistribusi. Kafka mengorganisasi pesan dalam *topic*, menyimpannya secara terurut dan terpartisi, serta mengizinkan banyak produser dan konsumer beroperasi secara paralel dalam *cluster* yang *fault tolerant [20]*.

Untuk mengelola format data yang dikirim melalui Kafka, ekosistem Kafka sering memanfaatkan Schema Registry, misalnya Confluent Schema Registry. Schema Registry menyediakan *serving layer* bagi *schema* (misalnya Avro, Protobuf) sehingga produser dan konsumer menyepakati struktur data yang konsisten berdasarkan ID schema yang tersimpan di *registry [21]*. Pendekatan ini mengurangi *coupling* antar layanan, mencegah *schema drift*, dan mempermudah *evolution* tipe data tanpa merusak kompatibilitas. Dalam arsitektur FMS, kombinasi Kafka + Schema Registry menjadi tulang punggung *Common Middleware Layer* yang menghubungkan IAS di kendaraan dan backend *modulith* di *cloud platform*.

2.2.11 Knex Query Builder dan Custom Migration Source

Knex.js adalah SQL *query builder* untuk Node.js yang mendukung berbagai DBMS seperti PostgreSQL, MySQL, dan SQLite. Knex menyediakan API berbasis *fluent interface* untuk menyusun pernyataan SQL SELECT/INSERT/UPDATE/DELETE secara programatik, sekaligus modul migrasi skema yang dapat dikelola dengan skrip versi [22].

Pada sistem migrasi, Knex memperbolehkan pengembang mengimplementasikan Custom Migration Source, yaitu objek dengan metode getMigrations, getMigrationName, dan getMigration yang bertugas menyediakan daftar dan isi fungsi migrasi. Mekanisme ini dapat dimanfaatkan untuk memetakan migrasi dari berbagai lokasi atau *package* modul. Dalam konteks FMS, konsep tersebut menjadi dasar pendekatan *schema per module*, di mana setiap modul memiliki folder migrasinya sendiri, tetapi pada saat *runtime runner* dapat menggabungkan dan menjalankan migrasi lintas modul secara terorkestrasi.

2.2.12 Monorepo dalam Lingkungan Pengembangan

Monorepo (*monolithic repository*) adalah pendekatan pengelolaan kode sumber di mana banyak proyek atau modul disimpan dalam satu repositori tunggal, dibandingkan dipisah menjadi beberapa *polyrepo*. Di industri, pendekatan ini digunakan oleh organisasi besar seperti Google dan Facebook untuk mempermudah koordinasi perubahan lintas modul, *refactoring global*, dan *code sharing* yang konsisten [23].

Monorepo sendiri memiliki keuntungan seperti semua modul dan dependensi dapat dilihat dan dikelola secara terpadu, perubahan arsitektural (misalnya perubahan *interface* modul) dapat dilakukan dan diuji dalam satu *commit*, serta *tooling* seperti *workspace* (npm/pnpm/bun) memungkinkan *build* dan *test* per modul tanpa kehilangan konsistensi repositori.

Untuk arsitektur *modular monolith* FMS, monorepo memudahkan penerapan kontrak modul yang seragam, code reuse antar modul, dan penyiapan *script* otomatis yang menghasilkan kerangka modul baru sesuai standar.

2.3 PENELITIAN TERKAIT

2.3.1 Enhancing Urban Electric Vehicle (EV) Fleet Management Efficiency in Smart Cities: A Predictive Hybrid Deep Learning Framework

Kerangka kerja hibrid deep learning untuk memprediksi beban pengisian baterai dan optimasi rute armada EV di kota pintar. Hasil penelitian ini menunjukkan akurasi prediksi mencapai 98,9% dan optimasi rute hybrid (Coati–NGO) secara signifikan mengurangi jarak perjalanan sehingga meningkatkan efisiensi infrastruktur pengisian serta menurunkan biaya operasional armada [24].

2.3.2 Fleet Management and Control System for Medium-Sized Cities Based in Intelligent Transportation Systems: From Review to Proposal in a City

Memodelkan sistem manajemen armada hibrida (EV dan kendaraan konvensional) untuk transportasi publik dengan tujuan mengoptimalkan konsumsi energi. Model berbasis orientasi objek yang mereka usulkan terbukti meningkatkan efisiensi operasional dan mengurangi konsumsi energi pada armada EV/hibrida [4].

2.3.3 Microservice-Architecture - a Practical Approach to Developing a Distributed Heterogeneous-Fleet-Management-Platform

Meneliti arsitektur mikroservis untuk sistem manajemen armada EV. Penelitian ini menunjukkan bahwa penerapan arsitektur mikroservis pada sistem armada kendaraan heterogen menghasilkan skalabilitas dan kemudahan pemeliharaan yang tinggi karena setiap layanan dapat dikembangkan dan dikelola secara independen [25].

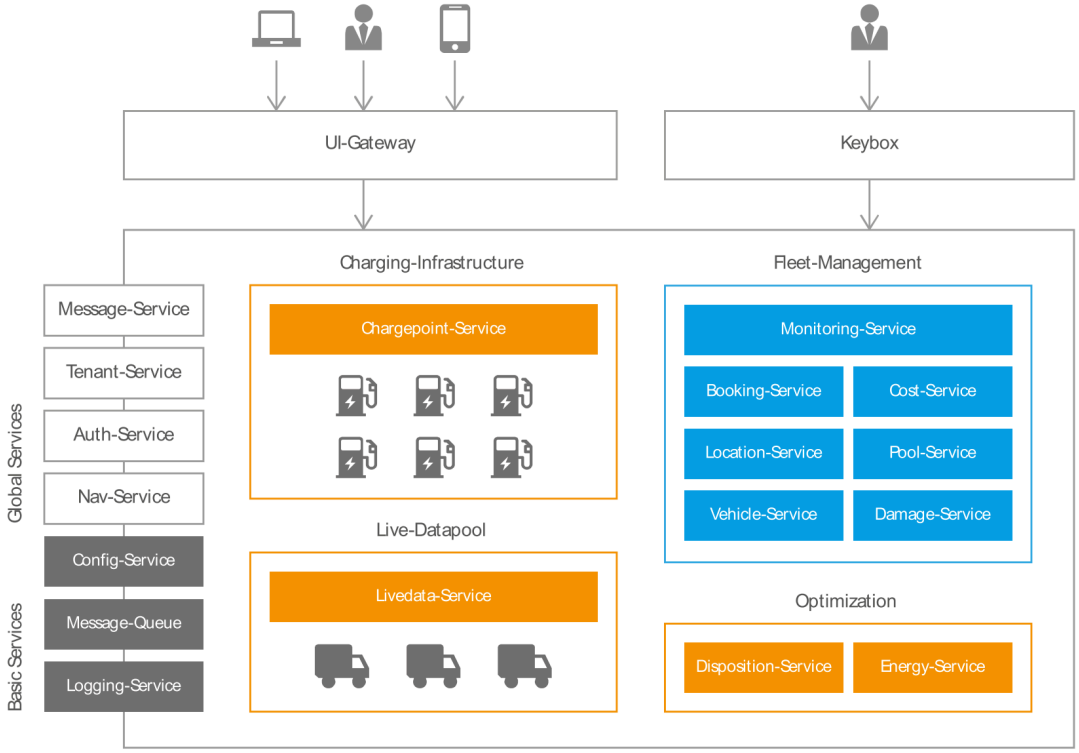


Figure 1: Coarse-grained namespaces in EMS

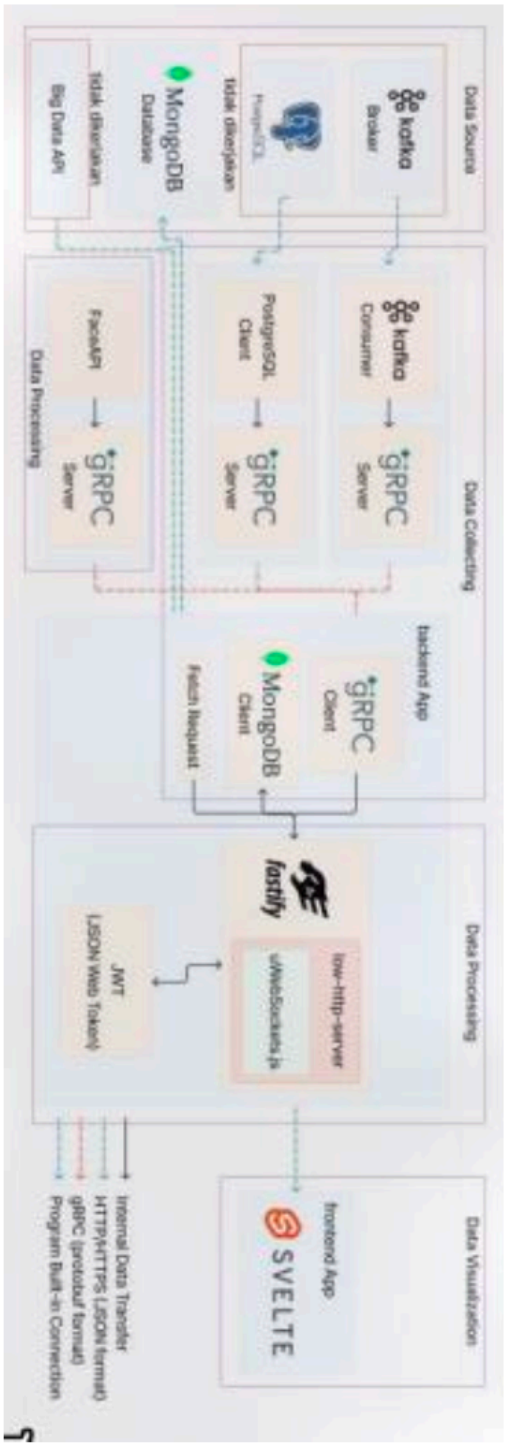
Gambar 2.4 Contoh Desain Sistem Fleet Management

2.3.4 Modular Monolith: Is This the Trend in Software Architecture?

Mengkaji pola arsitektur *Modular Monolith* sebagai alternatif antara monolitik tradisional dan mikroservis. Hasil kajian mereka mendefinisikan *modular monolith* sebagai pola arsitektur yang menggabungkan keunggulan monolitik dan mikroservis, dengan sistem dibagi menjadi modul-modul longgar terhubung yang masing-masing memiliki batasan jelas dan dapat digeser menjadi mikroservis secara terpisah jika diperlukan [8].

2.3.5 PENGEMBANGAN SERVER DAN USER INTERFACE SISTEM MANAJEMEN OPERASIONAL E-BUS

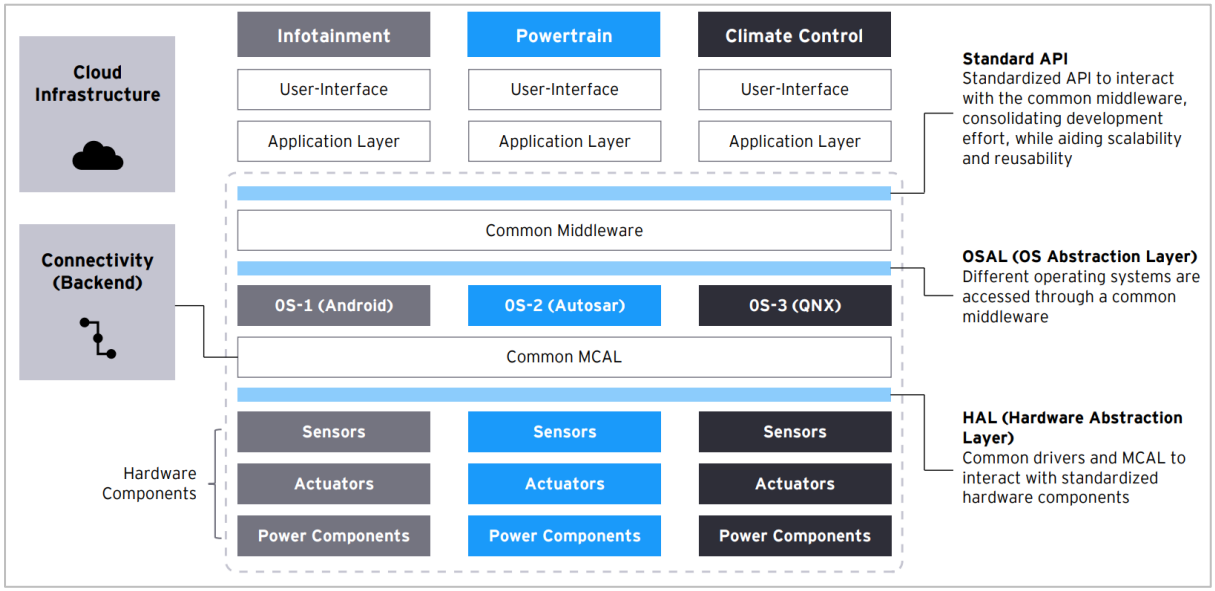
Proyek ini dilatarbelakangi oleh berbagai kekurangan pada transportasi umum bus, seperti kesulitan dalam estimasi waktu kedatangan yang tepat dan penentuan rute yang efisien, yang menyebabkan masyarakat enggan menggunakannya. Ketiadaan sistem manajemen yang menunjang operasional bus juga menyebabkan inefisiensi. Oleh karena itu, penelitian ini bertujuan untuk mengembangkan sistem manajemen operasional bus listrik (E-Bus) berbasis web yang dapat membantu baik dari sisi operasional maupun pelanggan. Sistem ini dirancang untuk menyediakan informasi lengkap mengenai kondisi bus di suatu kota dan mendukung pengelolaan operasional, serta menyediakan API untuk aplikasi pelanggan. Harapannya, sistem ini dapat mempermudah operasional dan pelanggan dalam mengandalkan transportasi umum bus. Sistem ini juga direncanakan untuk diimplementasikan secara langsung dalam lingkup produksi oleh PT. VKTR Indonesia [5].



Gambar 2.5 Desain Sistem Operasional Eksisting

2.3.6 Laporan EY Parthenon: The Software-Driven Revolution Redefining the Automotive Industry (2023)

Menurut laporan dari EY Parthenon India, industri otomotif saat ini sedang bergerak menuju paradigma *Software-Defined Vehicle* (SDV), yaitu pendekatan di mana fungsi kendaraan tidak lagi bergantung pada kombinasi *hardware-software* yang terikat dan tersebar pada ratusan ECU, tetapi digerakkan oleh platform software terpusat yang dapat diperbarui dan dikembangkan secara berkelanjutan. Transisi ini diperlukan karena arsitektur kendaraan tradisional dengan jaringan CAN terdistribusi, ECU domain-spesifik, serta ketergantungan kuat pada vendor Tier-1 tidak mampu lagi memenuhi tuntutan modern seperti konektivitas tinggi, integrasi layanan cloud, ADAS tingkat lanjut, hingga pengelolaan data telematika secara real-time. EY menggambarkan pergeseran ini sebagai revolusi menuju arsitektur yang lebih terpusat, modular, dan seragam, di mana layer software dipisahkan dari hardware melalui abstraction layer, API standar, dan platform komputasi berkinerja tinggi seperti yang ditunjukkan pada Gambar 2.6.



Gambar 2.6 Ekosistem SDV EY Parthenon

Dalam konteks SDV, data kendaraan dipandang sebagai aset strategis yang harus dapat diakses, diproses, dan dinormalisasi secara konsisten di seluruh ekosistem mobilitas, termasuk Fleet Management System (FMS). Karena itu, pendekatan *vendor-specific* dan fragmentasi format sinyal dianggap tidak lagi berkelanjutan. Pada Gambar 2.6 EY mengajukan pendekatan arsitektur dengan standarisasi struktur data, konsolidasi pipeline komunikasi, serta kemampuan untuk memanfaatkan data dari berbagai sensor dan protokol sebagai solusi dari kekurangan arsitektur kendaraan tradisional.

2.4 Posisi Penelitian

Berdasarkan tinjauan pustaka di atas, dapat dilihat bahwa proyek akhir ini menempati posisi unik sekaligus melengkapi berbagai penelitian terdahulu. Sebagian besar karya sebelumnya berfokus pada satu aspek spesifik dari permasalahan manajemen armada kendaraan listrik. Sebagai contoh, Lashin et al. (2024) berfokus pada optimasi performa dan efisiensi energi armada melalui model *hybrid*. Rojas et al. (2020) memberikan kerangka makro integrasi ITS untuk manajemen armada kota, namun belum masuk ke implementasi sistem nyata. Aziz (2023) telah membangun dasar sistem operasional *e-bus*, namun dengan keterbatasan arsitektur yang membuatnya kurang luwes untuk dikembangkan lebih lanjut.

Proyek akhir ini berbeda sekaligus mengisi celah (*gap*) di antara penelitian-penelitian tersebut. Tidak seperti Lashin yang berfokus pada aspek kendaraan hybrid, proyek ini berorientasi pada *software system* operasional dan layanan pengguna armada EV secara keseluruhan. Sementara karya Rojas bersifat konseptual, proyek ini merealisasikan implementasi *modular fleet management* dalam skala lebih kecil namun konkret. Dan yang terpenting, proyek ini menerapkan arsitektur *modular monolith*, sesuatu yang belum muncul dalam penelitian terdahulu di domain FMS EV. Pendekatan modulith memungkinkan sistem menggabungkan keunggulan solusi-solusi sebelumnya (integrasi data *real-time*, fitur manajemen lengkap, layanan pengguna) dalam satu *platform* utuh yang tetap mudah dikembangkan.

Dari sisi tren industri, proyek ini sejalan dengan arah transformasi SDV seperti diungkap EY Parthenon (2023), dengan mengadopsi prinsip standardisasi data dan sentralisasi platform di level FMS. Dengan demikian, kontribusi proyek akhir ini terletak pada inovasi arsitektur untuk FMS armada EV: menghadirkan sistem operasional+layanan yang komprehensif, fleksibel terhadap perubahan, serta siap terhubung dengan ekosistem kendaraan modern. Ini melengkapi karya-karya terdahulu dan memberikan pijakan bagi pengembangan lebih lanjut, misalnya menuju migrasi ke *microservices* ketika skala sistem membesar di masa datang, tanpa mengorbankan stabilitas yang telah dibangun. Proyek akhir ini diharapkan dapat menjembatani kesenjangan antara teori dan praktik dalam pengelolaan armada bus listrik, sekaligus menjadi contoh implementasi *modular monolith* di domain transportasi cerdas.

Halaman sengaja dikosongkan

BAB 3
DESAIN SISTEM

3.1 DESKRIPSI SOLUSI

Sistem yang diusulkan merupakan solusi terintegrasi untuk manajemen armada kendaraan listri. Arsitektur sistem ini mengikuti pola *modular monolith*, yaitu satu kesatuan aplikasi yang dibagi menjadi modul-modul fungsional dengan batasan yang jelas [8]. Dengan pendekatan ini, sistem memiliki keuntungan pengembangan cepat ala monolitik sekaligus modularitas komponen seperti pada mikroservis. Setiap modul saling berkomunikasi melalui antarmuka API. Pendekatan modular juga memudahkan pengembangan fitur baru atau migrasi ke arsitektur mikroservis di masa depan [8].

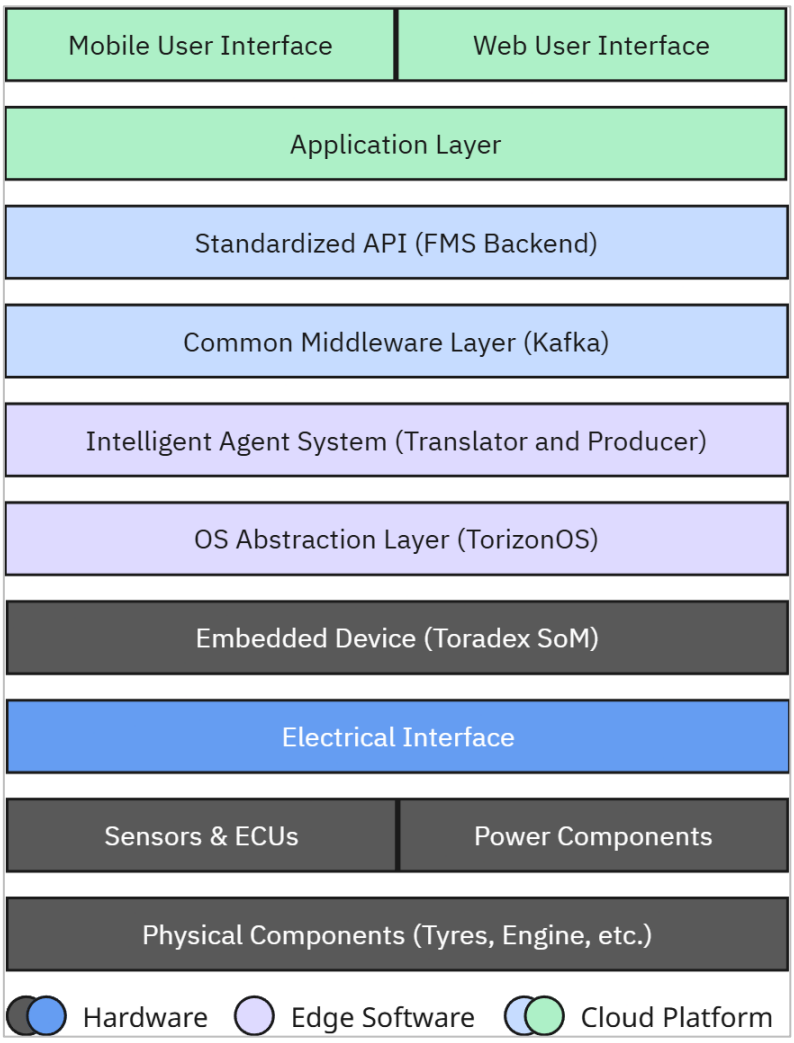
Implementasi *backend* menggunakan Fastify, sebuah framework web Node.js berperforma tinggi dan *overhead* rendah. Fastify dipilih karena kemampuannya menangani hingga puluhan ribu permintaan per detik dengan efisiensi sumber daya. Untuk penyimpanan data, digunakan kombinasi PostgreSQL dan MongoDB. PostgreSQL dipilih sebagai basis data relasional andal yang telah teruji selama puluhan tahun untuk menyimpan data terstruktur (seperti data pengguna dan jadwal). Sementara MongoDB digunakan sebagai basis data dokumen yang fleksibel dan mudah di-*scale*, mendukung skema dinamis untuk data telemetri kendaraan dan *log* sensor. Komunikasi dengan servis luar seperti client web atau *mobile* dikelola menggunakan protokol modern yakni sebagian API disajikan secara RESTful untuk kebutuhan antarmuka umum, Untuk data *real-time*, sistem memanfaatkan WebSocket dan Server-Sent Events (SSE). WebSocket digunakan untuk komunikasi dua arah yang interaktif (misalnya mobile app mengirimkan update lokasi user secara langsung), sedangkan SSE dipilih untuk aliran satu arah ke klien dengan *overhead* rendah (misalnya notifikasi jadwal baru, ETA kendaraan, *tracking* kendaraan) [26].

Komunikasi antar modul-modul internal pada proses *backend* dapat melalui dua cara. Untuk komunikasi asinkron, bisa menggunakan *shared* event-bus yang di-*provide* oleh module-runner. Hal ini secara mendukung pola *event-driven* dalam komunikasi antar modul. Untuk komunikasi langsung atau sinkron, suatu modul dapat melakukan *direct import* dari API yang diekspos oleh modul lainnya. API yang diekspos tiap modul merupakan suatu abstraksi (*interface*) dari fungsionalitas yang ada dalam modul tersebut, bukan implementasi langsung. Hal ini mendukung *type support* ketika menggunakan fungsionalitas antar modul tanpa melanggar batasan karena yang diekspos adalah suatu abstraksi.

Selain implementasi arsitektur *modular monolith,* diperlukan juga adanya kerangka kerja yang mendukung pengembangan berkelanjutan dengan pola *modular monolith* seperti perancangan development workflow dan dev-to-prod workflow yang menjelaskan bagaimana modul dikembangkan secara lokal (monorepo), diuji, lalu dipaketkan menjadi modul siap pakai untuk varian aplikasi yang berbeda.

3.1.1 Ekosistem Fleet Management System

Keseluruhan ekosistem pada Fleet Management System (FMS) dirancang untuk mengatasi fragmentasi antar domain kendaraan yang umum terjadi pada arsitektur tradisional maupun pada pendekatan *domain-silo* seperti yang diajukan dalam laporan EY pada bab 2.3.6. Dengan membangun rantai pemrosesan data yang terstandardisasi dari level sensor hingga aplikasi pengguna, ekosistem yang diajukan oleh proyek akhir ini pada Gambar 3.1 berupaya menyatukan berbagai sumber data kendaraan dalam satu jalur yang konsisten.



Gambar 3.1Ekosistem FMS

Pendekatan seperti pada Gambar 3.1 tidak hanya mendukung prinsip *Software-Defined Vehicle* (SDV), tetapi juga menyediakan struktur yang lebih modular, universal, dan mudah diintegrasikan oleh FMS. Secara umum, berikut adalah penjelasan per-komponen dari ekosistem FMS yang diajukan oleh proyek akhir ini.

Mobile UI dan Web UI

Lapisan ini berfungsi sebagai antarmuka bagi pengguna akhir seperti operator armada, teknisi, hingga *customer*. Seluruh informasi yang berasal dari kendaraan, seperti status operasional, telemetri, dan data lokasi, ditampilkan dalam format visual yang responsif baik melalui perangkat mobile maupun web. Kedua antarmuka menggunakan API yang sama.

Application Layer

Lapisan aplikasi mengelola logika operasional FMS seperti pemantauan kendaraan, manajemen armada, dan dashboard operasional. Semua fungsi di lapisan ini bekerja di atas API yang telah distandarisasi, sehingga tidak perlu mengetahui detail teknis terkait protokol sensor atau format data mentah dari ECU.

Standardized API (FMS Backend) – Bagian yang dikerjakan

Lapisan API berperan sebagai pintu masuk yang seragam bagi seluruh aplikasi dan layanan internal. API menyediakan akses yang telah dinormalisasi terhadap data telematika dan lokasi kendaraan. Dengan pendekatan ini, backend tidak lagi perlu menangani variasi format data sensor dari vendor yang berbeda, melainkan hanya berfokus untuk mengelola logika bisnis dengan berdasarkan pola arsitektur *modular monolith* serta membangun fungsionalitas yang dapat menunjang operasional manajemen armada.

Common Middleware (Kafka dan Database)

Lapisan middleware menyediakan infrastruktur komunikasi untuk menangani data streaming secara real-time dan penyimpanan terstruktur. Kafka berfungsi sebagai sarana distribusi data dari Intelligent Agent System (IAS) menuju backend, sedangkan database menyimpan data terstruktur untuk keperluan bisnis dan manajemen operasional armada. Lapisan ini menjadi tulang punggung pertukaran data antar komponen dalam ekosistem.

Intelligent Agent System (Translator dan Producer)

IAS merupakan komponen kunci dari ekosistem ini. Sistem ini bertanggung jawab membaca data sensor dari berbagai protokol seperti CAN J1939, OBD-II, DBC proprietary, maupun NMEA, kemudian melakukan decoding, pemetaan ke VSS, dan pembentukan payload Protobuf sebelum dipublikasikan ke middleware. IAS bertindak sebagai lapisan unifikasi yang menyatukan data lintas domain kendaraan menjadi format universal, sehingga mengubah data mentah menjadi informasi terstandardisasi.

OS Abstraction Layer (Torizon OS)

Torizon OS memberikan lapisan abstraksi perangkat lunak yang memudahkan deployment aplikasi, manajemen kontainer, serta komunikasi dengan perangkat keras secara stabil. Lapisan ini memungkinkan IAS bisa berjalan pada platform embedded.

Embedded Device (Toradex SoM)

Lapisan perangkat embedded berperan sebagai pusat pemrosesan di dalam kendaraan. Toradex SoM menerima sinyal dari sensor melalui berbagai antarmuka kelistrikan dan menjalankan Torizon OS serta IAS. Sistem ini menyediakan jembata pemrosesan untuk decoding CAN, parsing GPS, serta menjalankan pipeline konkuren pada IAS.

Electrical Interface

Lapisan ini mencakup konektivitas fisik seperti CAN bus, antarmuka serial, serta jalur komunikasi lain yang menghubungkan ECU dan sensor dengan perangkat embedded. Elemen inilah yang menjamin data dari domain-domain kendaraan dapat mencapai IAS secara stabil.

Sensors & ECUs, Power Components

Komponen ini mencakup sensor kendaraan, modul kontrol elektronik (ECU), unit baterai, motor traksi, serta perangkat kelistrikan lain yang menghasilkan data operasional kendaraan. Data mentah yang dihasilkan berbeda-beda format dan struktur tergantung vendor, sehingga memerlukan proses standardisasi di tingkat IAS.

Physical Components

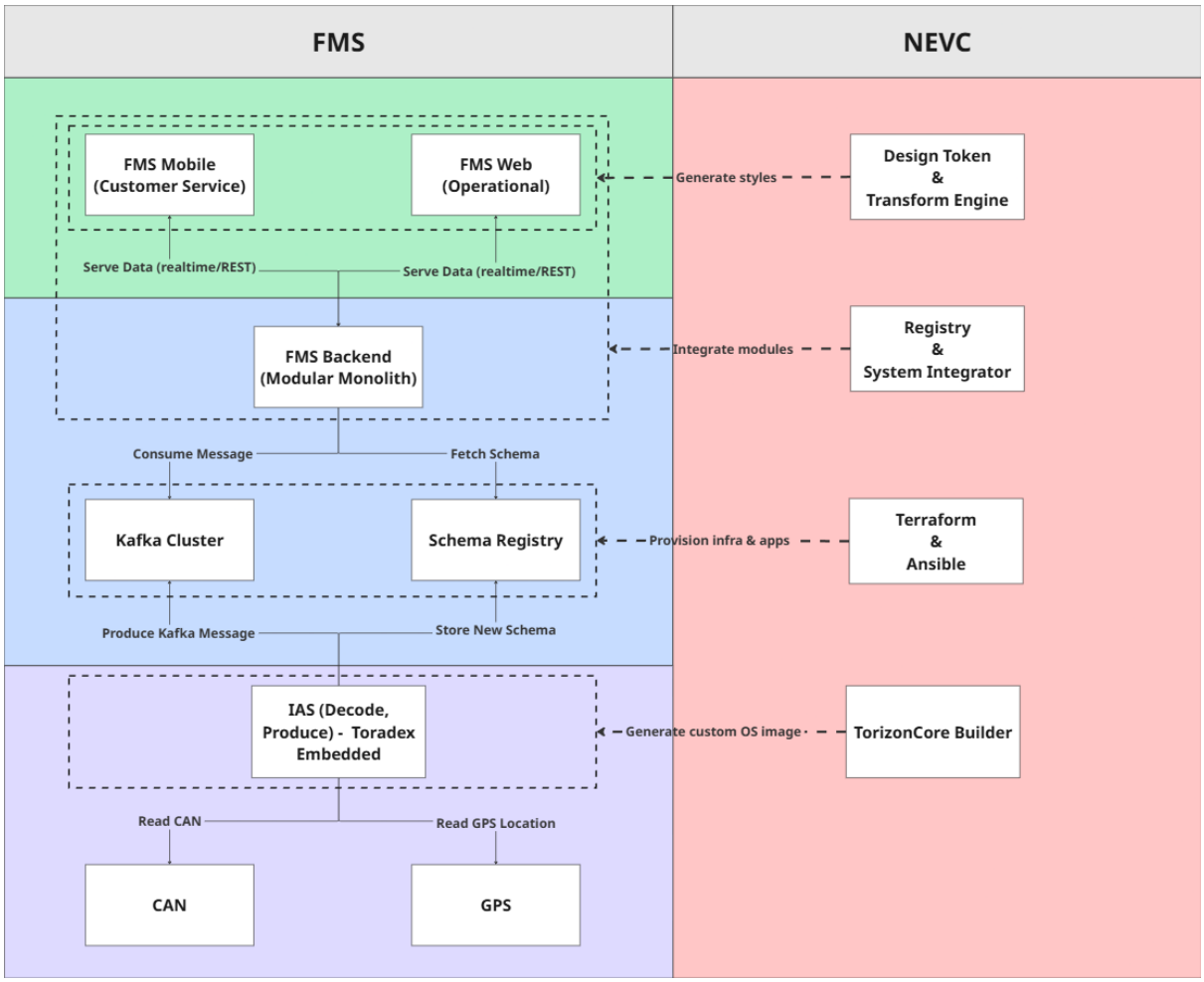
Lapisan ini mencakup komponen fisik seperti roda, motor listrik, sistem pengereman, dan struktur kendaraan lainnya yang menjadi sumber utama kondisi operasional yang dipantau melalui sensor.

Melalui struktur ekosistem ini, peran IAS dapat dipahami sebagai komponen *edge software* yang bertugas mengonversi data heterogen dari lapisan perangkat keras menjadi data telematika terstandarisasi yang siap digunakan oleh lapisan aplikasi dan antarmuka pengguna.

Secara keseluruhan, ekosistem ini menempatkan IAS sebagai titik konsolidasi utama yang menyatukan seluruh domain sensor kendaraan ke dalam jalur data yang standar dan terintegrasi. Berbeda dengan arsitektur yang diajukan pada laporan EY, yang masih mempertahankan struktur domain-silo di mana tiap subsistem kendaraan berkomunikasi secara terpisah sebelum mencapai cloud atau backend, pendekatan ini melakukan standarisasi lebih awal pada tingkat perangkat embedded. Dengan menormalkan data lintas domain di sumbernya, ekosistem yang diajukan pada proyek akhir ini menghilangkan fragmentasi struktural yang menjadi kelemahan arsitektur EY, sekaligus lebih sesuai dengan arah transisi menuju arsitektur *Software-Defined Vehicle* yang universal dan modular.

3.1.2 Alur Data Sistem

Alur data pada platform Fleet Management System (FMS) menggambarkan proses mulai dari pengambilan data sensor pada kendaraan hingga data tersebut ditampilkan kepada pengguna akhir melalui antarmuka aplikasi. Setiap komponen pada arsitektur memiliki peran spesifik dalam memastikan bahwa data yang diproses konsisten, terstandarisasi, dan dapat dimanfaatkan secara operasional oleh perusahaan pengguna FMS. Diagram alur data sistem ditunjukkan pada Gambar 3.2.



Gambar 3.2 Alur Data FMS dan Peran NEVC

Secara umum, alur data sistem terdiri dari tahap-tahap berikut:

Pengambilan Data Sensor Kendaraan

Data bersumber dari dua kategori utama, yaitu data CAN dari sistem elektronik kendaraan dan data NMEA melalui modul GPS. Data mentah ini bersifat heterogen, bergantung pada standar (seperti J1939, OBD-II, vendor-specific DBC) maupun pabrikan kendaraan.

Pemrosesan Data pada IAS

Intelligent Agent System (IAS) yang berjalan pada perangkat embedded (misalnya Toradex) menerima data mentah dari sensor. IAS melakukan fungsi utama meliputi decoding CAN/NMEA, pemetaan sinyal dictionary, pembentukan payload dengan format Protobuf, dan publikasi data ke Kafka. IAS berfungsi sebagai middleware penerjemah yang menyatukan berbagai format sensor menjadi format telematika yang seragam.

Distribusi Data melalui Kafka Cluster

Payload Protobuf yang diproduksi IAS dikirimkan ke Kafka Cluster, yang berperan sebagai *messaging backbone* yang memungkinkan data telematika dikonsumsi sistem pada lapisan lainnya secara real-time dan terdistribusi. Setiap pesan yang dikirim mengikuti skema Protobuf yang terdaftar dalam Schema Registry.

Manajemen Skema melalui Schema Registry

Schema Registry menyimpan dan mengelola versi skema Protobuf yang digunakan pada seluruh sistem. FMS Backend mengambil definisi skema dari komponen ini untuk memastikan kompatibilitas pemrosesan data.

Pemrosesan pada FMS Backend

FMS Backend berfungsi sebagai lapisan pengolah utama yang menerima data dari Kafka. Menjalankan proses seperti penyimpanan data, pengolahan telematika, normalisasi tambahan, dan integrasi antar modul fitur. Backend menerapkan arsitektur modular monolith yang memudahkan perluasan fitur berdasarkan kebutuhan pengguna.

Penyajian Data ke User Interface

Backend melayani data ke dua antarmuka utama, FMS Web untuk operator dan FMS Mobile untuk pengguna akhir. Data yang diambil dan diminta dilakukan melalui beberapa metode, secara *realtime* dengan protokol SSE atau WebSocket, dan melalui protokol RESTful API.

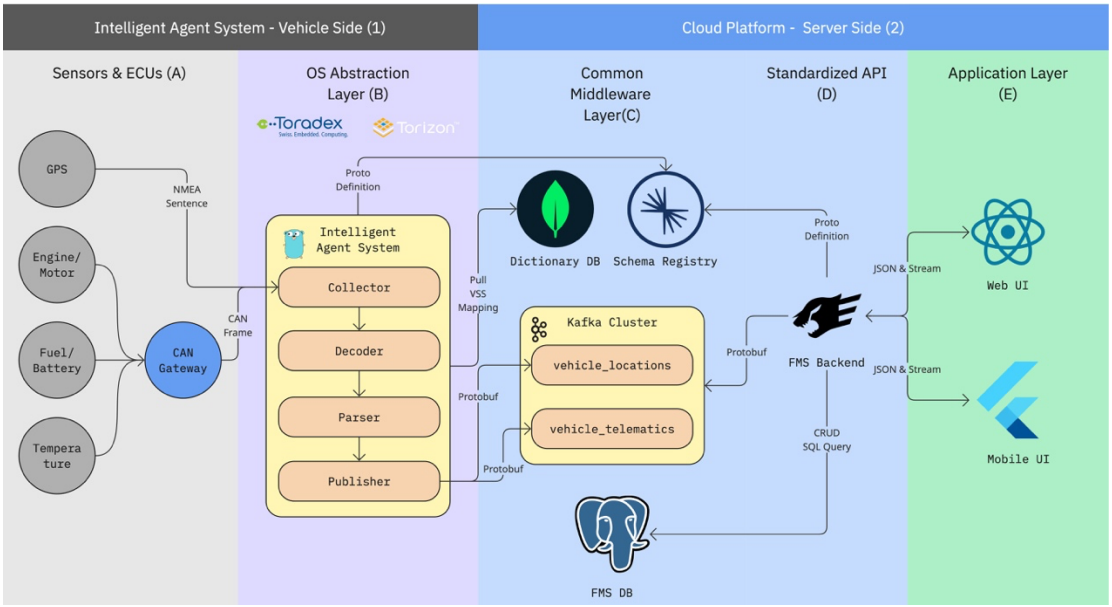
NEVC (System Integrator)

NEVC berperan dalam proses integrasi yang meliputi penyediaan dan provisioning perangkat embedded IAS, konfigurasi infrastruktur backend dan Kafka, integrasi modul-modul backend berdasarkan kebutuhan pelanggan, dan penyesuaian gaya tampilan antarmuka. Dengan demikian, NEVC memastikan bahwa FMS dapat disesuaikan dengan kondisi operasional dan lingkungan armada dari masing-masing pengguna.

Melalui alur data tersebut, sistem menyediakan proses yang terstruktur, dimulai dari pengambilan data sensor hingga penyajian informasi bagi pengguna akhir. IAS berperan sebagai komponen sentral dalam menjembatani keragaman format sensor kendaraan dan memastikan bahwa seluruh lapisan FMS menerima data dalam format yang konsisten dan terstandarisasi. Sehingga, fokus pengembangan *backend* dalam penelitian ini dapat dimaksimalkan dalam mengimplementasikan arsitektur *modular monolith* serta membangun modul-modul yang menunjang operasional dan layanan pengguna.

3.2 PERANCANGAN SISTEM

Proyek akhir ini berfokus pada salah satu bagian dari keseluruhan ekosistem FMS untuk bus listrik, yakni FMS *Backend*. Gambar 3.3 berikut merupakan representasi arsitektur menyeluruh dari ekosistem FMS yang dibangun.

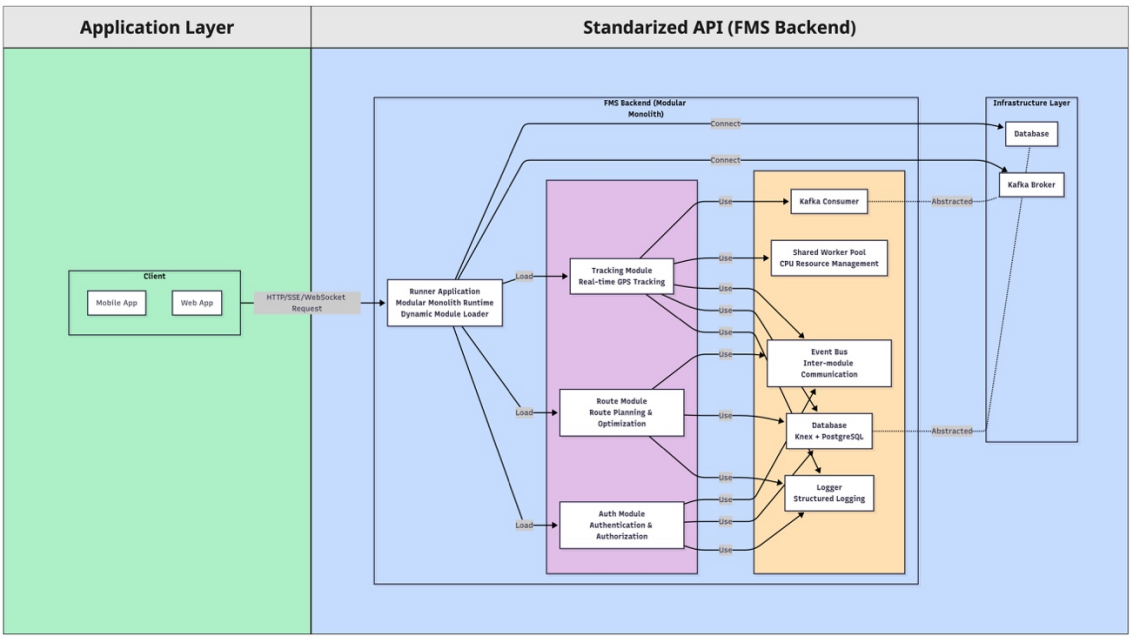


Gambar 3.3. Arsitektur Menyeluruh FMS

Arsitektur pada gambar 3.3 menunjukkan bahwa *backend* bertindak sebagai simpul tengah yang menjaga konsistensi data dan protokol sebelum menuju *end application*. Implementasi modul tracking yang sudah berjalan pada tahap progres ini menjadi bukti bahwa jalur data tersebut dapat direalisasikan secara *end-to-end*, sekaligus menjadi fondasi untuk pengembangan modul-modul berikutnya.

3.2.1 Desain Sistem FMS Backend Modular Monolith

Setelah melakukan studi literatur, berikut adalah hasil pengembangan desain arsitektur *backend* FMS pada proyek akhir ini.



Gambar 3.4. Arsitektur Backend FMS Modulith

Pada Gambar 3.4 dapat dilihat bahwa arsitektur *FMS Backend* secara garis besar terdiri dari tiga komponen utama yakni:

Application Layer (runner)

Bagian ini nantinya yang akan menjadi *entry point* utama dari *backend* yang memiliki tanggung jawab untuk melakukan *module bootstraping* dan *module loading*, menyiapkan Infrastructure Layer sebagai abstraksi dari koneksi dengan servis luar seperti kafka dan database yang akan digunakan oleh modul-modul yang di-load nantinya, serta *handling request* untuk diproses oleh modul-modul yang sudah di-load.

Modules Layer

Merupakan umpulan modul bisnis yang di-load secara dinamis oleh runner saat aplikasi mulai berjalan. Setiap modul harus mengimplementasikan antarmuka (interface) FmsModule pada entry point tiap modul agar bisa dijalankan oleh runner.

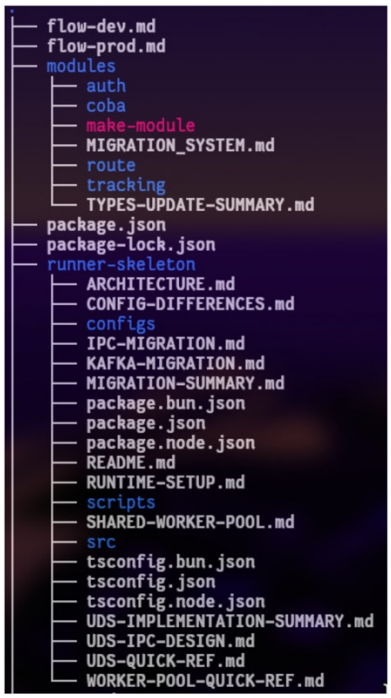
Infrastructure Layer

Infrastructure layer ini berisi shared objects yang nantinya akan digunakan oleh semua modul yang akan di-load oleh runner. Shared objects ini merupakan abstraksi beberapa fungsionalitas yang terkait dengan servis internal maupun external seperti logger, kafka, database, sharedworkerpool, serta shared event-bus.

Arsitektur menyeluruh backend ditunjukkan pada diagram sistem backend FMS yang dimuat pada Gambar 3.4. Diagram tersebut menampilkan bagaimana request dari klien masuk ke Fastify, diteruskan ke modul-modul bisnis, dan bagaimana modul-modul tersebut memanfaatkan lapisan infrastruktur yang sama.

3.2.2 Struktur Monorepo dan Modul

Pada fase pengembangan, implementasi backend disusun dalam bentuk monorepo yang berisi dua direktori utama:



Gambar 3.5 Struktur Monorepo

Runner-skeleton

Proyek Bun/Fastify yang berperan sebagai runner utama. Di dalamnya terdapat:

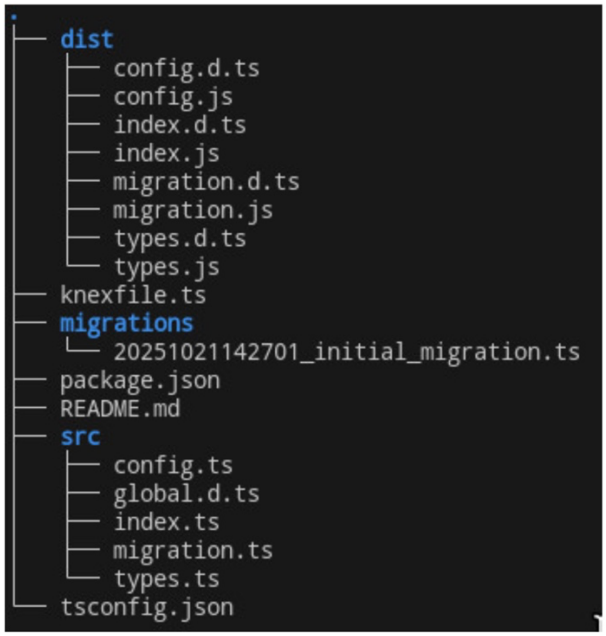
a. Berkas konfigurasi runtime (package.bun.json, tsconfig.*.json).
b. Berkas dokumentasi seperti ARCHITECTURE.md, RUNTIME-SETUP.md, dan ringkasan desain IPC serta worker pool.
c. Scripts untuk menyiapkan runtime Bun dan mengelola modul.

Modules

Direktori berisi modul-modul FMS, misalnya auth, tracking, dan route. Setiap modul mempunyai struktur yang seragam, dihasilkan oleh skrip “make-module”, yang antara lain berisi:

a. src/index.ts sebagai entry point modul yang mengimplementasikan interface “FmsModule”.
b. src/config.ts untuk mengelola konfigurasi modul dari environment pada proses utama (runner).
c. src/migration.ts sebagai custom migration source Knex.
d. Direktori migrations/ yang menyimpan berkas migrasi SQL/Knex untuk skema tiap modul.
e. Berkas knexfile.ts dan tsconfig.json lokal modul.

Struktur monorepo ini digambarkan dalam diagram hierarki folder yang menunjukkan relasi antara runner-skeleton/ dan modules/*.



Gambar 3.6 Contoh Struktur Module

Di dalam setiap modul, struktur proyek lebih rinci menunjukkan pemisahan antara kode sumber, konfigurasi, dan migrasi. Diagram khusus modul menampilkan bagaimana sebuah modul, misalnya auth, memiliki src/, dist/, dan migrations/ yang terorganisasi.

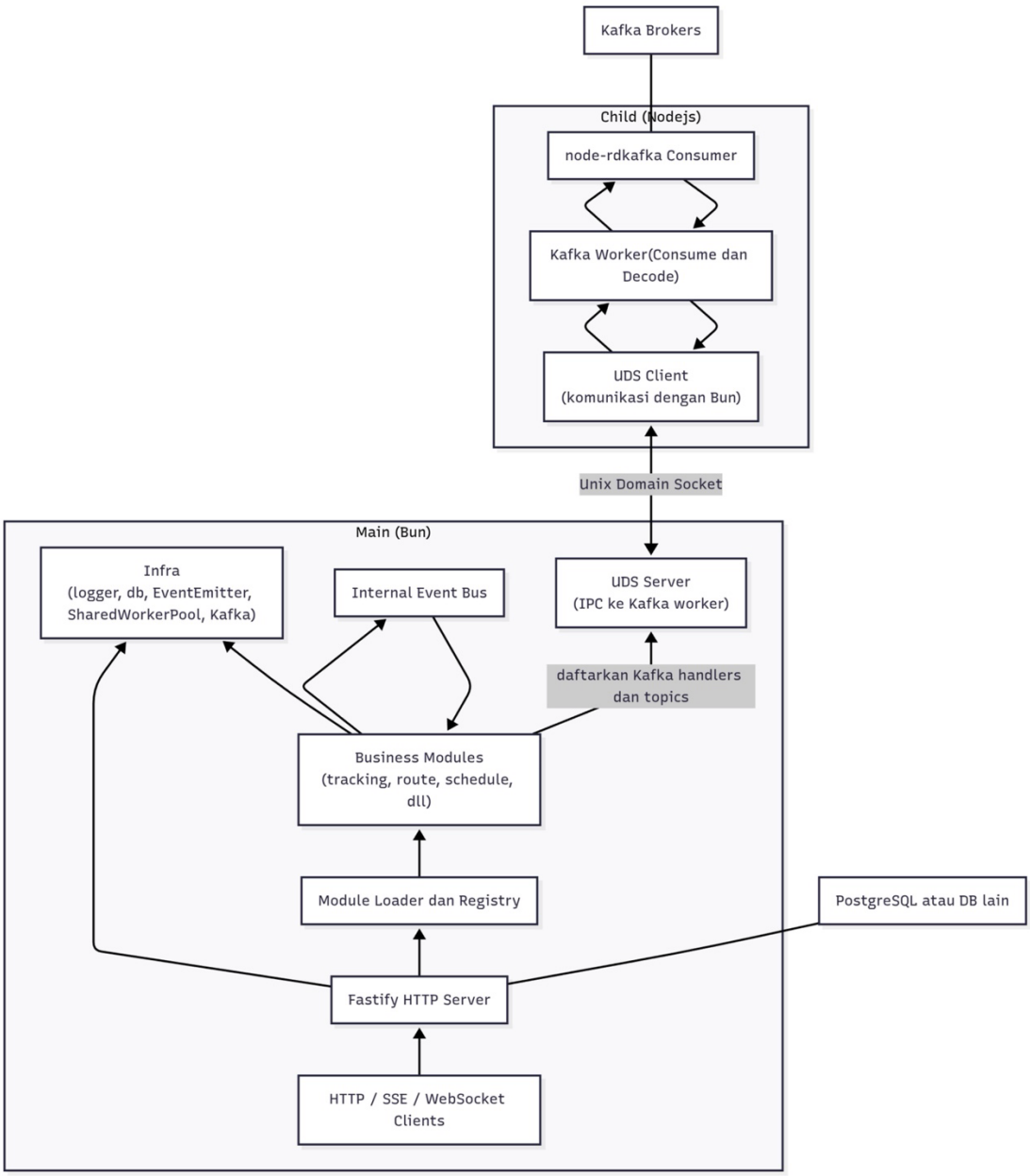
Agar modul dapat diaktifkan atau dinonaktifkan, runner menggunakan berkas konfigurasi configs/modules.json. Berkas ini memetakan identitas modul ke nama paket NPM (untuk skenario produksi) dan flag enabled. Contoh isinya antara lain:

```json
{
  "tracking": {
    "enabled": true,
    "packageName": "@ev-connect/tracking"
  },
  "auth": {
    "enabled": true,
    "packageName": "@ev-connect/auth"
  }
}
```

Format konfigurasi tersebut divisualisasikan dalam sebuah diagram yang menjelaskan cara runner membaca konfigurasi, mengimpor modul, dan menyusunnya berdasarkan dependensi sebelum dipanggil init dan register.

3.2.3 Arsitektur Multi-Proses Bun–Node

Untuk memanfaatkan keunggulan Bun sebagai runtime cepat bagi Fastify namun tetap menggunakan pustaka Kafka matang (node-rdkafka) yang berbasis Node.js, backend dirancang dengan arsitektur multi-proses. Proses utama menggunakan Bun, menjalankan server Fastify, modul-modul bisnis, event bus internal, dan manajemen worker pool. Proses anak (child process) menggunakan Node.js dan node-rdkafka untuk menangani konsumsi dan dekode pesan dari Kafka.



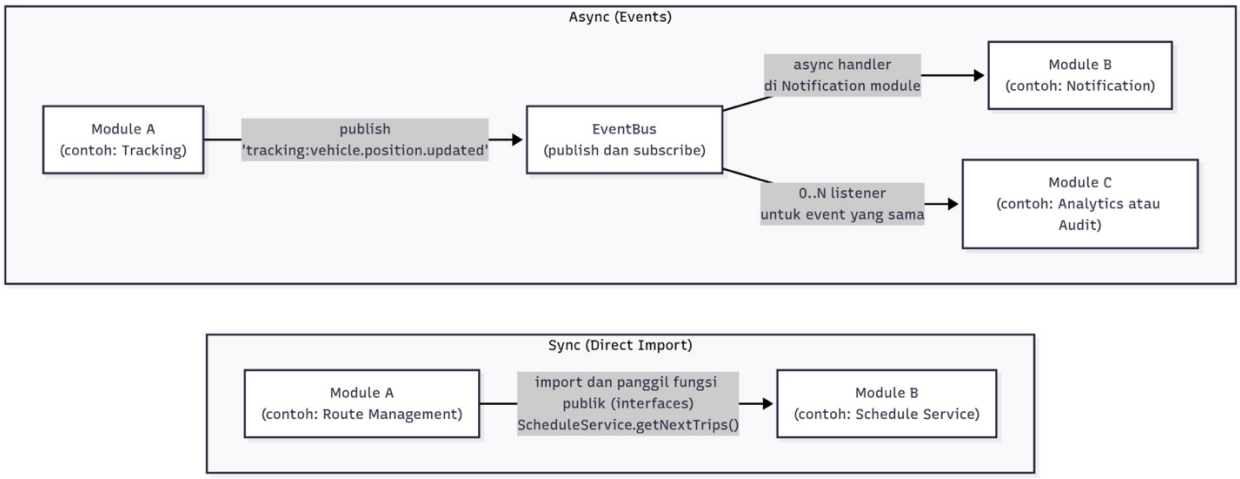
Gambar 3.7 Diagram Multi-process

Komunikasi antara kedua proses dilakukan menggunakan Unix Domain Socket (UDS). Alurnya adalah sebagai berikut:

1. Runner Bun menjalankan UDS server dan kemudian men-spawn proses anak Node.js.
2. Proses Node.js menginisialisasi node-rdkafka consumer, melakukan subscribe ke topik yang relevan, lalu mulai mengkonsumsi pesan dari Kafka.
3. Setiap pesan yang berhasil didekode dikirim ke proses Bun melalui UDS dalam format JSON ringkas.
4. Di sisi Bun, handler UDS meneruskan data tersebut ke modul terkait (misalnya modul tracking) melalui event bus atau langsung memanggil fungsi modul.

3.2.4 Pola komunikasi antar modul

Di dalam satu proses Bun, modul-modul FMS berinteraksi satu sama lain menggunakan dua pola utama:



Gambar 3.8 Komunikasi Antar Modul

1. Asinkron melalui event-bus terpusat dari runner.
   Modul yang menghasilkan event (misalnya modul tracking ketika posisi kendaraan diperbarui) akan memanggil eventBus.publish("tracking:vehicle.position.updated", payload). Modul lain dapat berlangganan event yang sama, contohnya modul notifikasi atau modul analitik yang ingin merekam statistik perjalanan.
2. Pola sinkron berbasis pemanggilan fungsi publik
   Untuk kasus di mana modul perlu mendapatkan data secara langsung, misalnya modul route membutuhkan jadwal keberangkatan, modul tersebut dapat mengimpor service publik dari modul lain (misalnya ScheduleService.getNextTrips()).

Kedua pola ini digambarkan dalam sebuah diagram yang memperlihatkan modul A (misalnya tracking) menerbitkan event ke event bus, kemudian dikonsumsi oleh modul B (notifikasi) dan modul C (analitik), serta contoh pemanggilan fungsi langsung antara modul route dan modul schedule.

Dengan memisahkan komunikasi asinkron dan sinkron, sistem dapat menjaga loose coupling antar modul sekaligus tetap memungkinkan koordinasi cepat ketika diperlukan.

3.2.5 Rancangan Migrasi Basis Data per Modul

Salah satu karakteristik penting backend ini adalah pendekatan schema per modul pada PostgreSQL. Setiap modul diberi skema tersendiri (misalnya auth, route, tracking), sehingga:

1. Tabel-tabel modul terisolasi dan tidak bercampur dengan modul lain.
2. Akses dan kebijakan backup dapat diatur per skema jika diperlukan.
3. Evolusi skema (penambahan kolom, tabel, dan sebagainya) dapat dikelola secara lebih terkontrol per modul.

```
interface IMigrationSource {
    getMigrations(): Promise<string[]>;
    getMigrationName(migration: string): string;
    getMigration(migration: string): Promise<{
        up: (knex: Knex) => Promise<void>;
        down: (knex: Knex) => Promise<void>;
        name: string;
    }>;
}
```

Gambar 3.9 Interface Custom Migration Source

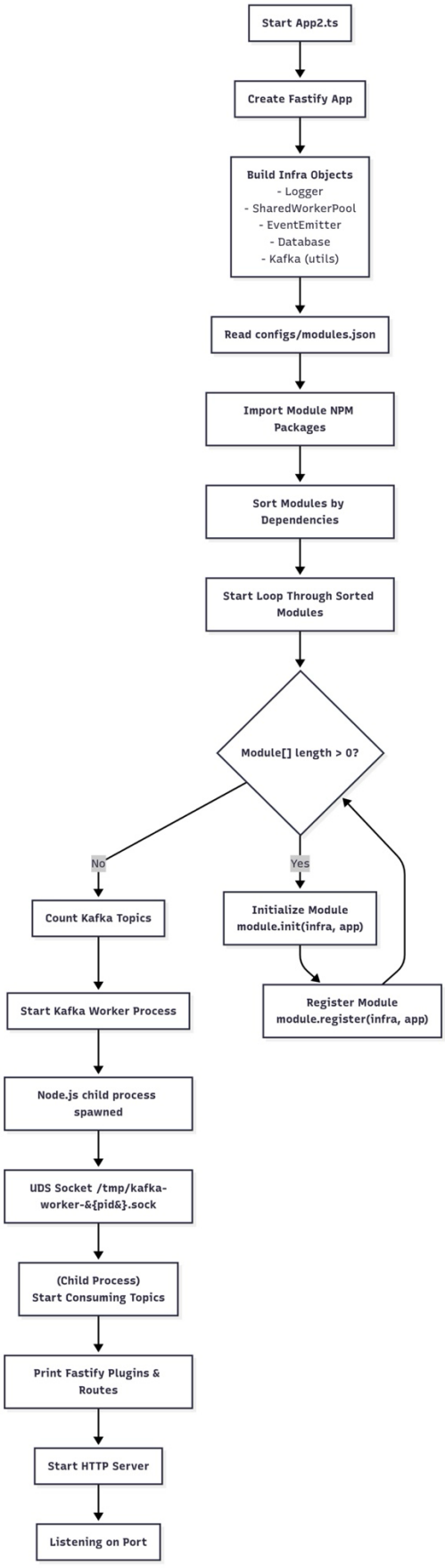
Pada tahap progres ini, pendekatan schema per modul telah diimplementasikan dan diuji pada modul auth dan route. Modul tracking belum memiliki migrasi karena fungsionalitas yang berjalan masih berfokus pada streaming data, namun modul tersebut tetap memiliki kerangka migration.ts sehingga dapat ditambahkan migrasi di tahap berikutnya.

3.2.6 Module Runner Lifecycle

Agar pengembang lain dapat mengikuti alur kerja aplikasi, dirancang sebuah siklus hidup (lifecycle) backend yang secara ringkas mencakup langkah-langkah berikut:

1. Runner membaca konfigurasi environment dan berkas modules.json.
2. Runner membangun objek infrastruktur (logger, koneksi database, *event bus*, *shared worker pool*, helper Kafka).
3. Runner mengimpor modul-modul yang diaktifkan dan menyusunnya berdasarkan dependensi.
4. Untuk setiap modul, runner memanggil fungsi init yang antara lain menjalankan migrasi basis data dan mendaftarkan plugin Fastify.
5. Setelah seluruh modul berhasil di-*init*, runner memanggil fungsi register untuk mendaftarkan *route*, *hook*, dan integrasi event.
6. Runner menyiapkan UDS server dan memulai *Kafka worker process*.
7. Setelah seluruh komponen siap, HTTP server Fastify mulai mendengarkan pada port yang telah ditentukan.

Urutan ini divisualisasikan dalam diagram alur yang menunjukkan blok-blok proses dan juga pengambilan keputusan (*decision*) seperti yang dapat dilihat pada Gambar 3.10.



Gambar 3.10 Module Runner Lifecycle

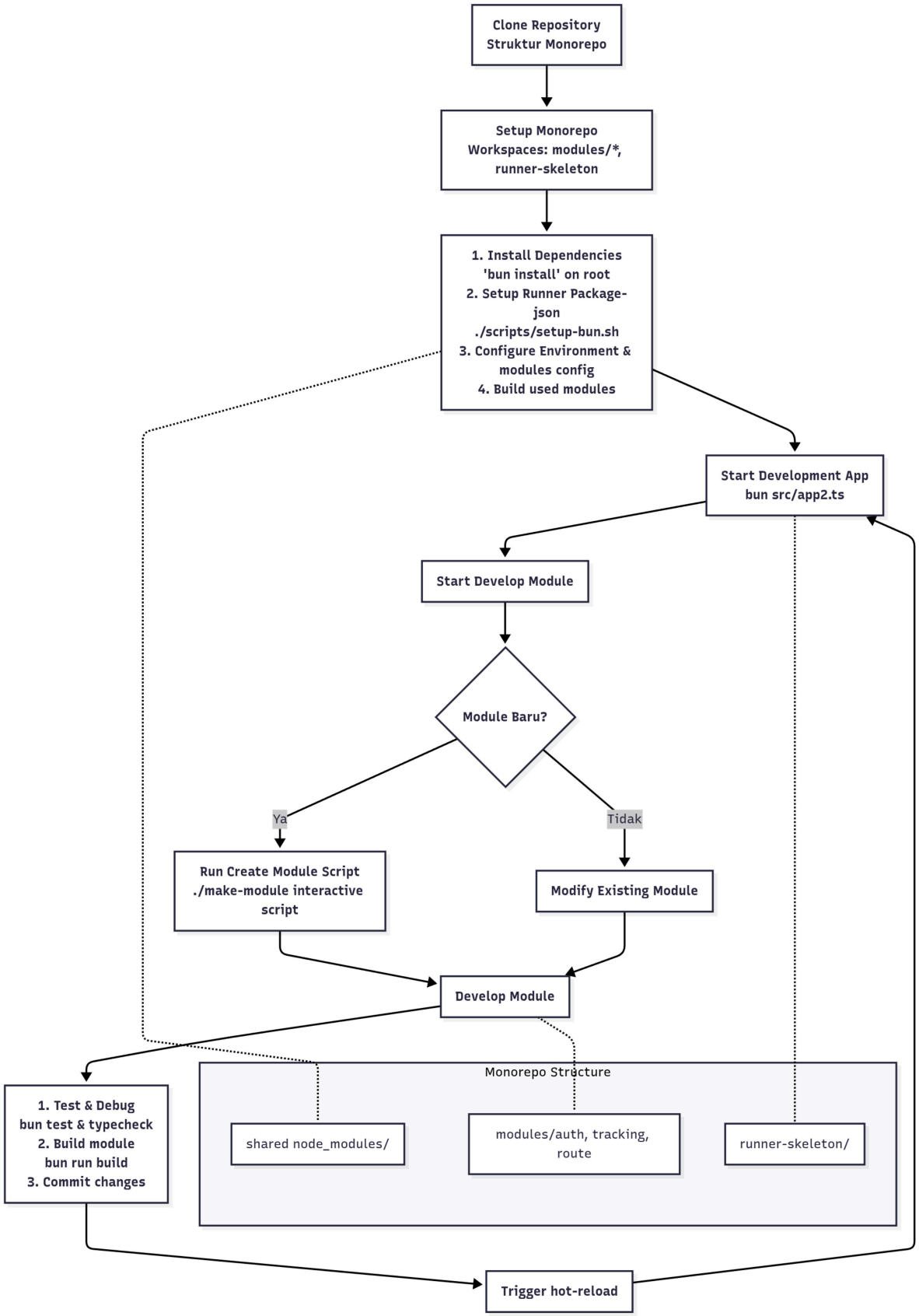
Dengan adanya dokumentasi lifecycle ini, proses onboarding bagi pengembang baru menjadi lebih mudah karena mereka dapat melihat dengan jelas pada tahap mana kode modul mereka akan dieksekusi.

3.2.7 Alur Kerja Fase Pengembangan

Untuk mendukung pengembangan modul yang konsisten, disusun sebuah development workflow yang memanfaatkan monorepo dan kemampuan hot reload Bun. Secara garis besar langkahnya adalah:

1. Mengkloning repositori monorepo yang berisi runner-skeleton/ dan direktori modules/.
2. Menjalankan bun install di root untuk mengunduh dependensi bersama.
3. Menjalankan skrip konfigurasi runtime, misalnya ./scripts/setup-bun.sh, untuk menyiapkan package.bun.json dan berkas konfigurasi lainnya.
4. Mengkonfigurasi environment dan configs/modules.json untuk memilih modul mana yang diaktifkan.
5. Memulai aplikasi pengembangan dengan bun src/app2.ts (atau skrip serupa) yang mendukung *hot reload*.
6. Untuk menambah modul baru, pengembang menjalankan skrip make-module yang akan membuat kerangka modul beserta berkas migrasinya.
7. Pengembang menulis kode modul, menjalankan *test* dan *typecheck*, lalu melakukan *build* modul jika diperlukan.

Urutan ini divisualisasikan dalam diagram yang menunjukkan jalur pengembangan modul baru maupun modifikasi modul yang sudah ada seperti yang dapat dilihat pada Gambar 3.11.



Gambar 3.11 Development Workflow

Workflow ini memastikan bahwa penambahan modul baru selalu mengikuti pola yang sama, sehingga meminimalkan risiko perbedaan standar antar modul.

3.2.8 Integrasi Alur Kerja Development dan Production

Selain alur pengembangan lokal, dirancang pula workflow produksi yang akan menjadi dasar bagi implementasi NEVC System Integrator di tahap berikutnya. Inti dari desain ini adalah memisahkan fase:

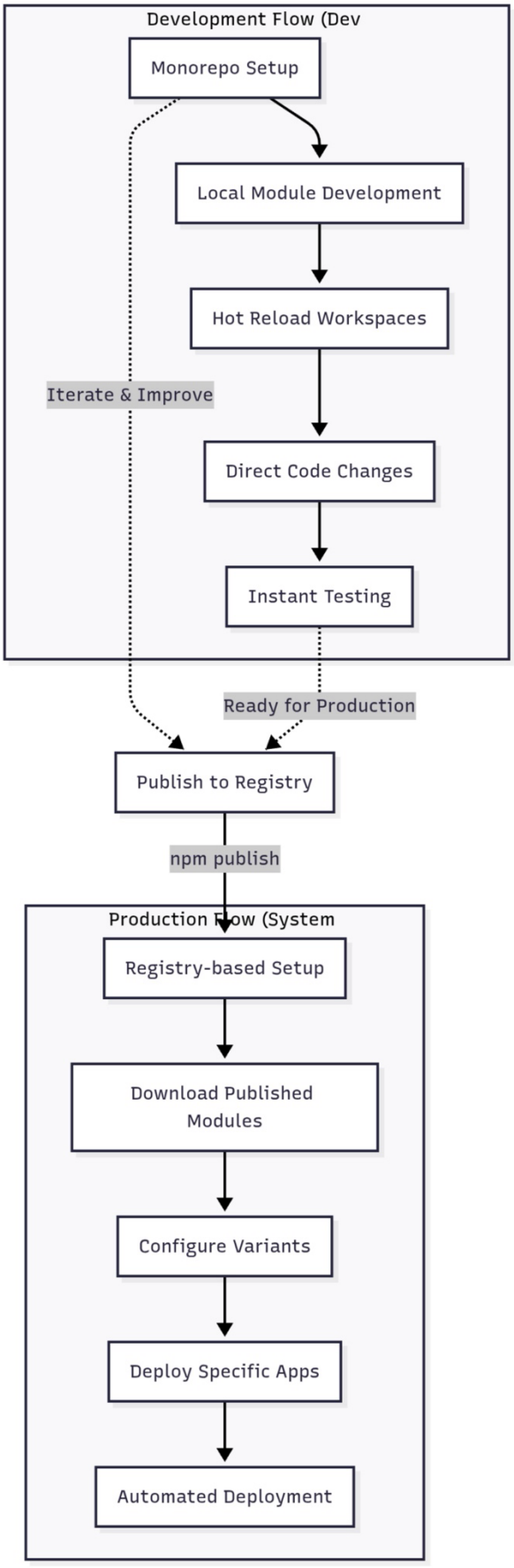
Fase Development

1. Modul dikembangkan dalam monorepo seperti dijelaskan pada Subbab 3.2.7.
2. Setelah fungsionalitas modul stabil, modul dapat di-build dan dipublikasi ke registry NPM privat dengan namespace, misalnya @ev-connect/*.

Fase Development-to-Production

1. NEVC (atau tim integrator) memilih kombinasi modul sesuai kebutuhan varian aplikasi (misalnya Tracking Only, Route Planning, atau Full Suite).
2. Runner skeleton dikloning sebagai basis aplikasi varian, kemudian configs/modules.json diisi sesuai modul yang dipilih.
3. Aplikasi varian menginstal modul-modul tersebut dari registry menggunakan bun install @ev-connect/....
4. Setelah environment produksi dikonfigurasi, aplikasi dijalankan dengan pengelolaan proses menggunakan PM2, systemd, atau Docker sesuai kebutuhan operator.

Diagram khusus dev-to-prod workflow yang mengaitkan secara langsung monorepo pengembangan dengan sistem berbasis registry di lingkungan produksi. Diagram ini memperlihatkan bahwa proses iterasi kode di monorepo, pengujian, dan build modul pada akhirnya bermuara pada perintah npm publish yang membuat modul tersedia bagi aplikasi varian di sisi operator.



Gambar 3.12 Dev-to-Prod Workflow

Desain ini menjadi fondasi bagi pengembangan NEVC System Integrator, yang kelak dapat mengotomatisasi pemilihan modul dan generasi konfigurasi aplikasi varian.

3.2.9 Rancangan Modul Contoh: Tracking, Auth, dan Route

Sebagai bagian dari progres pengerjaan, beberapa modul telah memiliki rancangan dan sebagian implementasi:

Modul Tracking

1. Fokus pada konsumsi data lokasi kendaraan dari Kafka dan distribusi ke klien melalui SSE.
2. Memanfaatkan jalur multi-proses Bun-Node dan SSE worker pool sehingga ratusan koneksi klien dapat dilayani tanpa mengganggu performa proses utama.
3. Rincian arsitektur modul tracking dan interaksinya dengan worker pool dijelaskan lebih lanjut dalam dokumentasi teknis dalam bentuk markdown file pada direktori modul.

Modul Auth

1. Menyediakan fungsionalitas autentikasi dan otorisasi pengguna.
2. Sudah memiliki skema basis data lengkap (tabel users, roles, user_roles) melalui migrasi di skema auth.
3. Entry point modul mencontohkan penggunaan fase init untuk memuat konfigurasi JWT dari environment, menjalankan migrasi, serta mendaftarkan plugin Fastify JWT dan fase register untuk mendefinisikan routing.

Modul Route

1. Dirancang untuk mengelola data rute dan jadwal operasional armada.
2. Pada tahap progres ini, modul route telah memiliki kerangka migrasi berbasis schema per modul, namun fungsionalitas API belum diselesaikan.

Keberadaan modul-modul contoh ini menjadi bukti bahwa desain modular monolith yang diusulkan dapat diimplementasikan secara konsisten di beberapa domain fungsional yang berbeda.

3.2.10 Ringkasan Progres Implementasi

Berdasarkan uraian di atas, progres utama proyek akhir pada saat penulisan laporan progres dapat diringkas sebagai berikut:

1. Desain arsitektur backend modular monolith telah terdokumentasi dan terealisasi dalam bentuk runner Bun/Fastify yang mampu memuat modul-modul bisnis secara dinamis.
2. Monorepo dan skrip make-module telah disiapkan sehingga proses pembuatan modul baru menjadi terstandarisasi.
3. Arsitektur multi-proses Bun–Node dengan UDS telah diimplementasikan dan diuji untuk jalur konsumsi Kafka, khususnya untuk topik posisi kendaraan.

4. Pola komunikasi antar modul (event bus dan pemanggilan fungsi langsung) telah dirancang dan disimulasikan melalui contoh skenario modul tracking dan modul pendukung lainnya.
5. Sistem migrasi basis data schema per modul telah diimplementasikan untuk modul auth dan route dengan custom migration source Knex.
6. Workflow pengembangan dan dev-to-prod sudah terdokumentasi dalam bentuk diagram yang jelas, menjadi dasar bagi implementasi otomatisasi dan integrasi dengan NEVC di tahap berikutnya.
7. Modul tracking telah berfungsi untuk melakukan streaming posisi kendaraan dari IAS melalui Kafka hingga ke klien menggunakan SSE, sehingga jalur data end-to-end untuk fitur pelacakan armada telah terverifikasi.

BAB 4
EKSPERIMEN DAN ANALISIS

Pada bab ini dilakukan serangkaian uji coba terhadap sistem yang telah dikembangkan. Pengujian ini bertujuan untuk memastikan bahwa implementasi sistem telah sesuai dengan desain yang dirumuskan pada bab sebelumnya, khususnya dalam implementasi arsitektur *modular monolith*, kerangka kerja yang mendukung pengembangan sistem modular, serta fungsionalitas modul-modul yang dikembangkan.

4.1 PARAMETER EKSPERIMEN

Pada tahap progres ini, eksperimen difokuskan pada validasi fungsi utama arsitektur *modular monolith* dan *workflow* pengembangan–produksi FMS Backend. Parameter utama eksperimen dirangkum sebagai berikut:

Jenis Pengujian

1. Pengujian fungsional (*functional test*) terhadap alur data modul *tracking* dari Kafka hingga klien (SSE).
2. Pengujian integrasi (*integration test*) antara *runner modular monolith*, modul bisnis (*tracking, auth, route*), sistem migrasi database menggunakan skema per modul, dan infrastruktur eksternal (*Kafka, Schema Registry, PostgreSQL, npm registry*).

Objek Pengujian

1. Aplikasi runner FMS *Backend* berbasis Bun–Fastify.
2. Modul *tracking* sebagai modul yang sudah fungsional penuh.
3. Modul *auth* dan *route* sebagai contoh modul dengan *schema per module* dan migrasi kustom.
4. Mekanisme *module loader* dan *sorting* dependensi berdasarkan *dependencies* di *meta* modul dan konfigurasi configs/modules.json.
5. *Workflow dev flow* (*monorepo*, *hot-reload*) dan *prod flow* (*install* modul dari npm *registry* internal).

Skenario uji

1. Verifikasi migrasi *database* per modul (*auth*, *route*).
2. Verifikasi konfigurasi modul dan dependensinya melalui modules.json.
3. Verifikasi alur data modul tracking (Kafka → Node worker → UDS → Bun runner → SSE client).
4. Verifikasi workflow pengembangan menggunakan monorepo.
5. Verifikasi workflow produksi dengan npm registry internal.

Lingkup Keberhasilan

1. Aplikasi backend dapat dijalankan tanpa error, modul dimuat sesuai konfigurasi, dan migrasi per modul berjalan/idempoten.
2. Pesan vehicle-location yang diproduksi ke Kafka dapat dikonsumsi worker, diteruskan ke runner, dan dikirim ke klien melalui SSE.
3. Dev flow dan prod flow yang dirancang dapat dieksekusi sesuai dokumentasi tanpa langkah improvisasi di luar skenario.

Batasan Eksperimen

1. Sumber data kendaraan belum berasal dari IAS aktual, tetapi disimulasikan dengan mem-produce pesan melalui Kafbat.
2. Pengujian tidak mencakup uji beban (throughput, latency, stress test), hanya fokus pada kebenaran fungsional dan integrasi.
3. Fungsional modul yang diuji penuh baru modul tracking; modul lain masih sebatas kerangka dan migrasi database.

4.2 KARAKTERISTIK DATA

Data yang digunakan dalam eksperimen ini merupakan data telematika kendaraan dengan fokus pada informasi posisi kendaraan. Pada lingkungan produksi, data ini akan dikirim oleh IAS di kendaraan dalam bentuk protobuf kemudian dikodekan ke Avro untuk dikirimkan melalui Kafka. Pada tahap progres ini, data disimulasikan menggunakan Kafka UI (Kafbat) dengan format Protobuf yang merepresentasikan struktur payload yang sama.

Secara umum, satu entri data lokasi kendaraan memuat atribut berikut:

```
{
        "vin": "vin PARJO",
        "lat": -7.278926,
        "lon": 112.790514,
        "timestamp": "7985112832142293000"
}
```

Pesan dikirim ke *topic* Kafka yang digunakan modul *tracking* (misalnya vehicle_locations) dan kemudian dikonsumsi oleh *worker* NodeJS melalui node-rdkafka. Worker ini meneruskan *payload* yang sudah didekodekan ke *runner* Bun melalui Unix Domain Socket (UDS) untuk di-*forward* ke klien melalui *endpoint* SSE.

Karakteristik data ini relatif ringan per pesan, tetapi dikirim secara kontinu. Oleh karena itu pada tahap progres ini pengujian hanya memverifikasi kebenaran alur data *end-to-end*, sementara karakteristik volume dan beban akan menjadi fokus pada tahap penelitian selanjutnya.

4.3 TEMPAT UJI COBA

Eksperimen dilaksanakan pada dua lingkungan utama:

1. **Lingkungan pengembangan lokal**
   - Laptop pengembang yang menjalankan runner FMS *Backend* (Bun–Fastify), modul bisnis (*tracking, auth, route*), serta klien SSE.
   - Digunakan untuk menjalankan *dev flow* di *monorepo*.
2. **Lingkungan infrastruktur bersama (*server VM*)**
   - Server yang menjalankan cluster Kafka, Schema Registry, dan PostgreSQL yang digunakan bersama tim proyek FMS–NEVC.
   - Server ini juga menjadi npm registry internal tempat modul FMS di-*publish* sebelum digunakan pada *prod flow*.

Topologi ini meniru kondisi produksi, di mana backend FMS akan berjalan dekat dengan infrastruktur messaging dan database, sementara pengembangan modul dilakukan secara lokal.

4.4 WAKTU UJI COBA

Seluruh proses pengembangan, eksperimen, dan uji coba pada proyek akhir ini dilaksanakan dalam rentang waktu bulan Agustus 2025 hingga Desember 2025. Pada periode tersebut dilakukan perancangan sistem, implementasi arsitektur, implementasi module fungsional, serta rangkaian pengujian yang diperlukan untuk mengevaluasi fungsionalitas dan kerangka kerja.

4.5 SPESIFIKASI PERALATAN UJI COBA

Karena fokus eksperimen adalah fungsionalitas, spesifikasi perangkat keras tidak menjadi faktor utama. Namun, untuk keperluan replikasi pengujian, spesifikasi peralatan yang digunakan dapat dirangkum sebagai berikut:

No	Perangkat	Ringkasan Spesifikasi
1	Laptop	CPU = 8 core RAM = 16 GB SSD = 512 GB OS = Arch Linux
2	VM Kafka	CPU = 4 core RAM = 12 GB Disk Size = 128 GB Running Kafka Broker, Kafka UI, Schema Registry
3	VM Database	CPU = 2 core RAM = 8 GB Disk Size = 128 GB Running postgresql
4	VM Gitlab	CPU = 4 core RAM = 12 GB Disk Size = 128 GB Running gitlab self-host

5	LXC FMS Staging	CPU = 2 core RAM = 1 GB Disk Size = 16 GB Running production FMS Backend

Spesifikasi ini sudah memadai untuk menjalankan uji fungsional yang dibutuhkan pada tahap progres.

4.6 HASIL EKSPERIMEN

4.6.1 Skenario Uji Coba

Berdasarkan parameter pada Subbab 4.1, dirancang lima skenario uji coba sebagai berikut.

Skenario A (Verifikasi migrasi database tiap modul)

1. Tujuan : Memastikan sistem migrasi Knex kustom dapat menjalankan migrasi per schema modul (auth dan route), bersifat idempoten, dan tercatat pada tabel historis migrasi.
2. Langkah utama:
    a. Mengaktifkan modul auth dan route pada modules.json.
    b. Menjalankan runner FMS Backend.
    c. Memeriksa log aplikasi dan tabel migrasi untuk memastikan migrasi dijalankan dan tidak diulang saat aplikasi dijalankan kembali.

Skenario B (Konfigurasi modul dan dependensi)

1. Tujuan : Memastikan module loader membaca modules.json, melakukan penyortiran modul berdasarkan dependencies, serta menampilkan warning jika terdapat dependensi yang belum diaktifkan.
2. Variasi :
    a. B1: Mengaktifkan hanya modul tracking yang memiliki dependensi ke modul auth, diharapkan muncul warning missing dependency: auth.
    b. B2: Mengaktifkan modul auth dan tracking, diharapkan modul auth diinisialisasi lebih dahulu, tidak ada warning dependensi.

Skenario C (Alur fungsional modul tracking)

1. Tujuan : Memastikan alur end-to-end modul tracking berjalan dari Kafka hingga klien SSE.
2. Langkah utama:
    a. Menjalankan Kafka worker NodeJS dan runner Bun.
    b. Mengirim pesan vehicle-location melalui Kafka UI (Kafbat) ke topic yang digunakan modul tracking.
    c. Mengakses endpoint SSE di backend dan memverifikasi bahwa data lokasi yang diterima sesuai dengan pesan yang dikirim.

Skenario D (Workflow pengembangan (dev flow) di monorepo)

1. Tujuan : Memastikan struktur monorepo dan skrip make-module mendukung pembuatan modul baru dan modifikasi modul.
2. Langkah utama:
    a. Mengkloning monorepo dan menjalankan bun install pada root.
    b. Menjalankan skrip pembuatan modul untuk membuat modul contoh.
    c. Menjalankan aplikasi dengan Bun, melakukan perubahan pada kode modul, dan mengamati perubahan yang segera tercermin pada hasil eksekusi tanpa publish package.

Skenario E (Workflow produksi (prod flow) dengan npm registry internal)

1. Tujuan : Memastikan modul backend FMS dapat dipaketkan sebagai paket npm scoped (@ev-connect/...), dipublish ke registry internal, dan diinstal pada runner skeleton untuk suatu varian aplikasi.
2. Langkah utama:
    a. Melakukan build modul dan publish ke npm registry internal.
    b. Mengkonfigurasi .npmrc dan configs/modules.json pada runner skeleton varian tertentu.
    c. Menjalankan bun install dan menjalankan aplikasi untuk memverifikasi bahwa modul yang dipasang dari registry dapat diload dan berfungsi.

Dokumentasi rinci langkah dan log tiap skenario telah direkap sebagai lampiran.

4.6.2 Hasil Uji Coba

Berdasarkan pelaksanaan lima skenario tersebut, hasil pengujian dapat dirangkum sebagai berikut:

Hasil Uji Coba Skenario A

1. Saat aplikasi dijalankan dengan modul auth dan route aktif, log menunjukkan bahwa sistem menjalankan migrasi untuk masing-masing modul dan schema.
2. Ketika aplikasi dijalankan kembali pada database yang sama, tidak ada migrasi yang diulang; log menggambarkan status “up to date” untuk modul yang sudah pernah dimigrasi. Hal ini menunjukkan bahwa mekanisme idempotensi migrasi per modul berjalan dengan baik.
3. Tabel historis migrasi di schema auth dan route berisi entri sesuai script migrasi yang disediakan.

Hasil Uji Coba Skenario B

1. Pada variasi B1 (hanya modul tracking yang diaktifkan), aplikasi tetap dapat berjalan tetapi log menampilkan warning yang menjelaskan bahwa modul tracking memiliki dependensi pada modul auth yang belum diaktifkan. Ini menunjukkan bahwa deteksi dependensi modul berjalan sesuai desain.

2. Pada variasi B2 (auth dan tracking diaktifkan), modul auth diinisialisasi lebih dahulu dan tidak muncul warning dependensi. Urutan inisialisasi modul pada log sesuai hasil penyortiran berdasarkan daftar dependencies di meta modul.

Hasil Uji Coba Skenario C

1. Worker Kafka berbasis node-rdkafka berhasil terhubung ke cluster Kafka dan berlangganan topic lokasi kendaraan.
2. Pesan vehicle-location yang diproduksi melalui Kafbat berhasil dikonsumsi worker, diteruskan melalui UDS ke runner Bun, dan dikirimkan ke klien melalui endpoint SSE.
3. Pada sisi klien, stream SSE menampilkan pembaruan posisi kendaraan sesuai data yang dikirim, menandakan bahwa alur data end-to-end modul tracking telah berfungsi walaupun sumber datanya masih berupa simulasi.

Hasil Uji Coba Skenario D

1. Struktur monorepo memungkinkan pemisahan jelas antara runner skeleton dan direktori modules/*. Skrip make-module berhasil menghasilkan kerangka modul baru dengan struktur standar (entry point, konfigurasi, dan folder migrasi).
2. Setelah modul dibuat atau dimodifikasi, menjalankan aplikasi pada mode pengembangan menunjukkan bahwa perubahan segera tercermin tanpa perlu konfigurasi tambahan yang rumit. Ini mendukung tujuan untuk mempermudah pengembangan modul oleh anggota tim yang berbeda.

Hasil Uji Coba Skenario E

1. Modul yang telah dibangun dapat dipublish ke npm registry internal dengan nama scoped (@ev-connect/tracking, @ev-connect/auth, dan sebagainya).
2. Runner skeleton varian produksi dapat menginstal modul tersebut melalui bun install dengan konfigurasi .npmrc dan configs/modules.json yang tepat.
3. Saat aplikasi dijalankan, log menunjukkan bahwa modul yang di-install dari registry dapat diinisialisasi dan diregistrasi sama seperti modul lokal. Hal ini menunjukkan bahwa packaging modul untuk kebutuhan multi-variant deployment telah berjalan sesuai rancangan.

Secara keseluruhan, seluruh skenario uji coba yang direncanakan pada tahap progres ini berhasil dieksekusi dan memenuhi ekspektasi fungsional.

4.7 ANALISIS HASIL EKSPERIMEN

Berdasarkan hasil eksperimen, beberapa poin analisis dapat diambil sebagai berikut:

Validasi arsitektur modular monolith

Pengujian terhadap migrasi per modul, konfigurasi modul, dan urutan inisialisasi menunjukkan bahwa arsitektur modular monolith yang diusulkan dapat mengekspresikan batas modul secara eksplisit. Setiap modul memiliki schema database dan migrasi sendiri, serta metadata dependensi yang dibaca oleh module loader. Hal ini mendukung tujuan arsitektural untuk menjaga kemandirian modul dan mempermudah pengelolaan variasi sistem.

Kelayakan alur data tracking end-to-end

Walaupun belum terkoneksi ke IAS aktual, alur data simulasi dari Kafka hingga SSE menunjukkan bahwa backend FMS sudah siap menerima stream lokasi kendaraan dari middleware bersama. Ketika IAS telah stabil, penyesuaian yang diperlukan relatif kecil pada sisi *producer* di kendaraan, karena kontrak data di Kafka sudah tervalidasi di sisi backend.

Efektivitas workflow pengembangan–produksi

Dev flow berbasis monorepo dan skrip make-module terbukti mempermudah pembuatan modul baru dan eksperimen arsitektural tanpa mengganggu modul lain. Sementara itu, prod flow berbasis npm registry internal memungkinkan modul dipaketkan dan digunakan kembali pada berbagai varian aplikasi FMS. Kombinasi keduanya menunjukkan bahwa tujuan untuk menyusun workflow yang mendukung multi-variant delivery sudah tercapai pada tahap progres ini.

Keterbatasan implementasi modul fungsional

Dari sisi fungsional bisnis, baru modul tracking yang teruji secara end-to-end. Modul auth dan route baru diuji pada level migrasi dan arsitektur, belum sampai integrasi dengan klien. Dengan demikian, tujuan proyek terkait kelengkapan fitur operasional FMS baru tercapai sebagian, dan akan menjadi fokus pada kelanjutan proyek akhir.

Keterbatasan jenis pengujian

Eksperimen yang dilakukan masih terbatas pada pengujian fungsional dan integrasi pada skala kecil. Belum ada pengujian beban, pengujian *fault tolerance*, maupun evaluasi non-fungsional lain seperti konsumsi sumber daya dan *scalability*. Hal ini perlu diperluas pada tahap penelitian berikutnya ketika seluruh modul utama FMS sudah selesai.

Dari analisis di atas dapat disimpulkan bahwa rancangan arsitektur dan workflow backend FMS modular monolith sudah tervalidasi secara fungsional, namun masih diperlukan pengembangan modul tambahan dan pengujian lanjutan untuk mencapai seluruh tujuan proyek akhir.

BAB 5
PENUTUP

5.1 KESIMPULAN

Berdasarkan rangkaian perancangan, implementasi, dan eksperimen yang telah dilakukan, dapat disimpulkan bahwa pengembangan Sistem Manajemen Armada untuk kendaraan listrik berbasis arsitektur modular monolith telah mencapai sebagian besar tujuan penelitian. Dari sisi arsitektur, backend FMS berhasil diwujudkan sebagai sebuah modular monolith dengan pemisahan modul bisnis yang jelas, kontrak antarmuka modul yang seragam, serta dukungan infrastruktur yang terintegrasi dengan Kafka, Schema Registry, dan PostgreSQL. Penerapan fase init dan register pada setiap modul, penggunaan schema per module melalui mekanisme migrasi Knex kustom, serta keberadaan module loader yang mampu membaca metadata dan menyortir dependensi menunjukkan bahwa rancangan arsitektur tidak berhenti pada tingkat konseptual, tetapi benar-benar terimplementasi dan dapat dijalankan.

Pengujian fungsional yang difokuskan pada modul tracking membuktikan bahwa alur data telematika dari Kafka hingga klien dapat berjalan secara end-to-end. Walaupun sumber datanya masih berupa simulasi dari Kafka UI, hasil uji coba menunjukkan bahwa worker NodeJS yang menggunakan node-rdkafka mampu mengonsumsi pesan vehicle-location, meneruskannya ke runner Bun melalui Unix Domain Socket, dan akhirnya menyajikan data tersebut kepada klien melalui SSE. Hal ini mengindikasikan bahwa backend yang dibangun sudah siap menerima integrasi IAS aktual di tahap berikutnya, karena kontrak data dan jalur distribusi pesan di sisi server telah tervalidasi.

Di sisi lain, penelitian ini juga berhasil merancang dan membuktikan kelayakan workflow pengembangan dan produksi yang selaras dengan kebutuhan multi-tim dan multi-varian. Struktur monorepo yang memisahkan runner skeleton dan direktori modules/*, ditambah skrip make-module, mempermudah pengembang untuk membuat dan memodifikasi modul tanpa konfigurasi berulang. Pada jalur produksi, kemampuan memaketkan modul sebagai scoped package dan mempublikasikannya ke npm registry internal memungkinkan varian aplikasi FMS dibentuk melalui kombinasi modul yang berbeda tanpa perlu menyalin kode. Hasil uji coba menunjukkan bahwa modul yang diinstall dari registry dapat diload dan diregistrasi seperti modul lokal, sehingga tujuan untuk menyediakan kerangka kerja pengembangan–produksi yang mendukung multi-variant delivery dapat dinyatakan tercapai pada tahap progres ini.

Meskipun demikian, dari sisi fungsional bisnis, capaian penelitian masih bersifat parsial. Modul tracking telah berfungsi penuh dan teruji, namun modul lain seperti auth dan route baru sampai pada tahap kerangka arsitektur dan migrasi database, belum memberikan layanan operasional secara end-to-end kepada pengguna. Dengan demikian, tujuan penelitian yang berkaitan dengan kelengkapan fungsi operasional FMS baru terpenuhi sebagian dan

masih memerlukan penyempurnaan pada fase lanjutan. Secara keseluruhan, penelitian ini telah membuktikan bahwa pendekatan arsitektur modular monolith dan workflow pengembangan–produksi yang diusulkan adalah layak dan operasional untuk FMS Backend, sekaligus membuka ruang kerja yang jelas untuk penyelesaian modul-modul berikutnya dan pengujian non-fungsional pada tahap penelitian selanjutnya.

5.2 SARAN

Berdasarkan keterbatasan yang ditemui pada tahap progres ini, beberapa saran yang dapat diajukan untuk pengembangan lanjutan adalah sebagai berikut:

Penyelesaian dan pengujian modul fungsional lainnya

Modul auth, route, dan modul operasi lainnya perlu diselesaikan dan diuji hingga tingkat integrasi penuh dengan klien web dan mobile. Pengujian harus mencakup skenario bisnis nyata seperti autentikasi pengguna operator, perencanaan rute, dan pengelolaan jadwal..

Integration testing dengan IAS pasa perangkat embedded (Toradex)

Ketika IAS telah stabil, perlu dilakukan pengujian end-to-end mulai dari sensor dan ECU di kendaraan, IAS, Kafka, FMS Backend, hingga antarmuka pengguna. Pengujian ini akan memvalidasi kompatibilitas protokol dan reliabilitas sistem pada kondisi operasional sebenarnya.

Pengujian non-fungsional dan scalability

Tahap berikutnya perlu mencakup pengujian performa (latensi, throughput), ketahanan terhadap kegagalan (fault tolerance), serta kemampuan scaling ketika jumlah kendaraan dan pesan telematika meningkat. Hasil pengujian tersebut penting untuk menilai kelayakan FMS sebagai sistem yang siap produksi.

Penguatan aspek observabilitas dan operational excellence

Untuk mendukung pengoperasian jangka panjang, backend FMS perlu dilengkapi dengan mekanisme *logging* terstruktur, *metrics* (misalnya dengan Prometheus), dan *alerting* yang terintegrasi dengan workflow DevOps. Hal ini akan mempermudah tim NEVC System Integrator dalam memantau kesehatan layanan.

Dokumentasi dan standarisasi modul

Seiring bertambahnya modul, dokumentasi teknis dan kontrak antarmuka modul perlu distandarkan agar modul yang dikembangkan oleh tim berbeda tetap konsisten. Dokumentasi ini juga penting untuk mendukung pengembangan ekosistem modul FMS di masa depan.

Dengan menyelesaikan saran-saran tersebut, diharapkan proyek akhir ini tidak hanya memenuhi kebutuhan akademik, tetapi juga memberikan kontribusi nyata terhadap pengembangan ekosistem Fleet Management System untuk kendaraan listrik yang fleksibel, terukur, dan siap diindustrialisasi.

DAFTAR PUSTAKA

[1] H. M. Sistig, P. Sinhuber, M. Rogge, dan D. U. Sauer, "Evaluating costs and operations of public bus fleet electrification," *npj Sustainable Mobility and Transport 2025 2:1*, vol. 2, no. 1, hlm. 1–14, Apr 2025, doi: 10.1038/s44333-025-00030-y.

[2] M. Farahpoor, O. Esparza, dan M. Soriano, "Comprehensive IoT-driven Fleet Management System for Industrial Vehicles," *IEEE Access*, 2023, doi: 10.1109/ACCESS.2023.3343920.

[3] Kemenhub, "Kemenhub Terus Dorong Digitalisasi di Sektor Transportasi - Dishub." Diakses: 9 Desember 2025. [Daring]. Tersedia pada: https://dishub.acehprov.go.id/2024/03/09/kemenhub-terus-dorong-digitalisasi-di-sektor-transportasi/

[4] B. Rojas, C. Bolaños, R. Salazar-Cabrera, G. Ramírez-González, Á. P. de la Cruz, dan J. M. M. Molina, "Fleet management and control system for medium-sized cities based in intelligent transportation systems: From review to proposal in a city," *Electronics (Switzerland)*, vol. 9, no. 9, hlm. 1–25, Sep 2020, doi: 10.3390/ELECTRONICS9091383.

[5] Farid Abdul Aziz, "PENGEMBANGAN SERVER DAN USER INTERFACE SISTEM MANAJEMEN OPERASIONAL E-BUS," 2023.

[6] S. Mohammed, J. Fiaidhi, dan M. Tang, "Towards using Microservices for Transportation Management: The New TMS Development Trend," Feb 2020, doi: 10.36227/TECHRXIV.11728965.V1.

[7] "(PDF) Telematics and Big Data: Revolutionizing Fleet Management in the Automotive Industry." Diakses: 29 Juni 2025. [Daring]. Tersedia pada: https://www.researchgate.net/publication/386112807_Telematics_and_Big_Data_Revolutionizing_Fleet_Management_in_the_Automotive_Industry

[8] R. Su dan X. Li, "Modular Monolith: Is This the Trend in Software Architecture?," 2024, Diakses: 29 Juni 2025. [Daring]. Tersedia pada: https://doi.org/xx.xxx/xxx_x

[9] "When (modular) monolith is the better way to build software | Thoughtworks United States." Diakses: 29 Juni 2025. [Daring]. Tersedia pada: https://www.thoughtworks.com/en-us/insights/blog/microservices/modular-monolith-better-way-build-software

[10] "Fast and low overhead web framework, for Node.js | Fastify." Diakses: 29 Juni 2025. [Daring]. Tersedia pada: https://fastify.dev/

[11] "Polyglot Persistence: Why It's Essential for Modern Applications | by Anh Trần Tuấn | tuanhdotnet | Medium." Diakses: 29 Juni 2025. [Daring]. Tersedia pada: https://medium.com/tuanhdotnet/polyglot-persistence-why-its-essential-for-modern-applications-83d9b1681a53

[12] "PostgreSQL: About." Diakses: 29 Juni 2025. [Daring]. Tersedia pada: https://www.postgresql.org/about/

[13] "Why Use MongoDB And When To Use It? | MongoDB." Diakses: 29 Juni 2025. [Daring]. Tersedia pada: https://www.mongodb.com/resources/products/fundamentals/why-use-mongodb

[14] “REST - Wikipedia.” Diakses: 29 Juni 2025. [Daring]. Tersedia pada: https://en.wikipedia.org/wiki/REST
[15] “Server-sent events - Web APIs | MDN.” Diakses: 29 Juni 2025. [Daring]. Tersedia pada: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events
[16] “WebSocket - Wikipedia.” Diakses: 29 Juni 2025. [Daring]. Tersedia pada: https://en.wikipedia.org/wiki/WebSocket
[17] Nodejs, “Child process | Node.js v25.2.1 Documentation.” Diakses: 10 Desember 2025. [Daring]. Tersedia pada: https://nodejs.org/api/child_process.html?utm_source=chatgpt.com#class-childprocess
[18] Blizzard, “GitHub - Blizzard/node-rdkafka: Node.js bindings for librdkafka.” Diakses: 10 Desember 2025. [Daring]. Tersedia pada: https://github.com/Blizzard/node-rdkafka?utm_source=chatgpt.com
[19] “unix(7) - Linux manual page.” Diakses: 10 Desember 2025. [Daring]. Tersedia pada: https://man7.org/linux/man-pages/man7/unix.7.html
[20] Apache, “Apache Kafka.” Diakses: 10 Desember 2025. [Daring]. Tersedia pada: https://kafka.apache.org/
[21] Confluent, “Schema Registry for Confluent Platform | Confluent Documentation.” Diakses: 10 Desember 2025. [Daring]. Tersedia pada: https://docs.confluent.io/platform/current/schema-registry/index.html
[22] “SQL Query Builder for Javascript | Knex.js.” Diakses: 10 Desember 2025. [Daring]. Tersedia pada: https://knexjs.org/
[23] “Monorepo Explained.” Diakses: 10 Desember 2025. [Daring]. Tersedia pada: https://monorepo.tools/
[24] M. Aldossary, “Enhancing Urban Electric Vehicle (EV) Fleet Management Efficiency in Smart Cities: A Predictive Hybrid Deep Learning Framework,” *Smart Cities 2024, Vol. 7, Pages 3678-3704*, vol. 7, no. 6, hlm. 3678–3704, Des 2024, doi: 10.3390/SMARTCITIES7060142.
[25] I. Hoffmann, J. Ostermann, dan K. Sivarasah, “Microservice-Architecture-a Practical Approach to Developing a Distributed Heterogeneous-Fleet-Management-Platform”.
[26] “Why Server-Sent Events (SSE) are ideal for Real-Time Updates.” Diakses: 29 Juni 2025. [Daring]. Tersedia pada: https://talent500.com/blog/server-sent-events-real-time-updates/

Lampiran 1

Skenario A

1. Siapkan .env untuk koneksi database

```
# Database Configuration
DB_HOST=localhost
DB_PORT=5432
DB_USERNAME=fms-admin
DB_PASSWORD=fms-admin-password
DB_NAME=fms
```

2. Aktifkan Satu Modul yang sudah implementasi custom migration source (contoh. auth)

```
{
   "tracking": {
      "enabled": false,
      "packageName": "@ev-connect/tracking"
   },
   "route": {
      "enabled": fale,
      "packageName": "@ev-connect/route"
   },
   "auth": {
      "enabled": true,
      "packageName": "@ev-connect/auth"
   }
}
```

3. Jalankan runner dan amati log untuk memastikan migrasi berjalan dan aplikasi berhasil memasuki tahap LISTENING

```
$ bun --watch src/app2.ts
[08:13:55 UTC] INFO: Module added
   module: "auth"
[08:13:55 UTC] INFO: Resolved module load order
   order: [
     "auth"
   ]
[08:13:55 UTC] INFO: Initializing module
   module: "auth"
[08:13:55 UTC] INFO: Running module migrations
   module: "auth"
```

```
    folder: "/home/sultan/PA/fms/modules/auth/migrations"
[08:13:55 UTC] INFO: No pending migrations
    module: "auth"
[08:13:55 UTC] INFO: Fastify JWT plugin registered
[08:13:55 UTC] INFO: Module initialized successfully
    module: "auth"
[08:13:55 UTC] INFO: Initialized
    module: "auth"
    ms: 30
[08:13:55 UTC] INFO: Registering module routes and hooks
    module: "auth"
[08:13:55 UTC] INFO: Module routes and hooks registered successfully
    module: "auth"
[08:13:55 UTC] INFO: Registered
    module: "auth"
    ms: 1
[08:13:55 UTC] INFO: All modules loaded, initialized, and registered
    count: 1
[08:13:55 UTC] WARN: No Kafka topics registered, skipping worker startup
[08:13:55 UTC] INFO: Server listening at http://127.0.0.1:3000
[08:13:55 UTC] INFO: Server listening at http://172.16.0.154:3000
[08:13:55 UTC] INFO: Server listening at http://172.16.0.27:3000
[08:13:55 UTC] INFO: Server listening at http://172.22.0.1:3000
[08:13:55 UTC] INFO: └── (empty root node)
    ├── /
    │   ├── health (GET, HEAD)
    │   └── a
    │       ├── dmin/workers/stats (GET, HEAD)
    │       └── uth/migrations/status (GET, HEAD)
    └── * (OPTIONS)
[08:13:55 UTC] INFO: Server listening
    port: 3000
```

4. Akses endpoint `/tracking/migrations/status` untuk memastikan status migrasi modul tracking

```
$ curl -s http://localhost:3000/auth/migrations/status | jq
{
  "module": "auth",
  "migrationsCompleted": 1,
  "migrationsPending": 0,
  "status": "up-to-date"
}
```

Lampiran 2

Skenario B

Case 1 (tracking, dependensi warning)

1. Ubah config modules.json untuk enable modul tracking saja.
   $ cat configs/modules.json

```
{
  "tracking": {
    "enabled": true,
    "packageName": "@ev-connect/tracking"
  },
  "route": {
    "enabled": false,
    "packageName": "@ev-connect/route"
  },
  "auth": {
    "enabled": false,
    "packageName": "@ev-connect/auth"
  }
}
```

2. Start runner dan amati log untuk memastikan module added, module load dengan benar sesuai dependensi, dan runner menampilkan log warning apabila dependencies tidak terpenuhi (modul disabled, tidak dapat ditemukan, tidak di-define dalam config)

```
$ bun --watch src/app2.ts
[08:29:20 UTC] INFO: Module added
    module: "tracking"
[08:29:20 UTC] WARN: Dependency missing
    module: "tracking"
    missing: "auth"
[08:29:20 UTC] INFO: Resolved module load order
    order: [
      "tracking"
    ]
```

Case 2 (tracking + auth, dependensi penuh, load benar)

1. Ubah config modules.json untuk enable modul tracking saja.
   $ cat configs/modules.json

```
{
  "tracking": {
    "enabled": true,
    "packageName": "@ev-connect/tracking"
  },
  "route": {
```

```
    "enabled": false,
    "packageName": "@ev-connect/route"
  },
  "auth": {
    "enabled": true,
    "packageName": "@ev-connect/auth"
  }
}
```

2. Start runner dan amati log untuk memastikan module added, module load dengan benar sesuai dependensi, dan runner menampilkan log warning apabila dependencies tidak terpenuhi (modul disabled, tidak dapat ditemukan, tidak di-define dalam config)

```
$ bun --watch src/app2.ts
[08:31:59 UTC] INFO: Module added
  module: "tracking"
[08:31:59 UTC] INFO: Module added
  module: "auth"
[08:31:59 UTC] INFO: Resolved module load order
  order: [
    "auth",
    "tracking"
  ]
```

Lampiran 3

Skenario C

1. Siapkan .env untuk koneksi kafka broker dan schema registry

```
# Kafka Configuration
KAFKA_BROKER_URL=10.10.11.4:19092,10.10.11.4:19094,10.10.11.4:19096
KAFKA_TOPIC_VEHICLE_LOCATION=vehicle-location
KAFKA_TOPIC_VEHICLE_OBD=vehicle-obd
KAFKA_CLIENT_ID=fms-backend
KAFKA_GROUP_ID=fms-backend-group

# Schema Registry Configuration
SCHEMA_REGISTRY_URL=http://10.10.11.4:8085
```

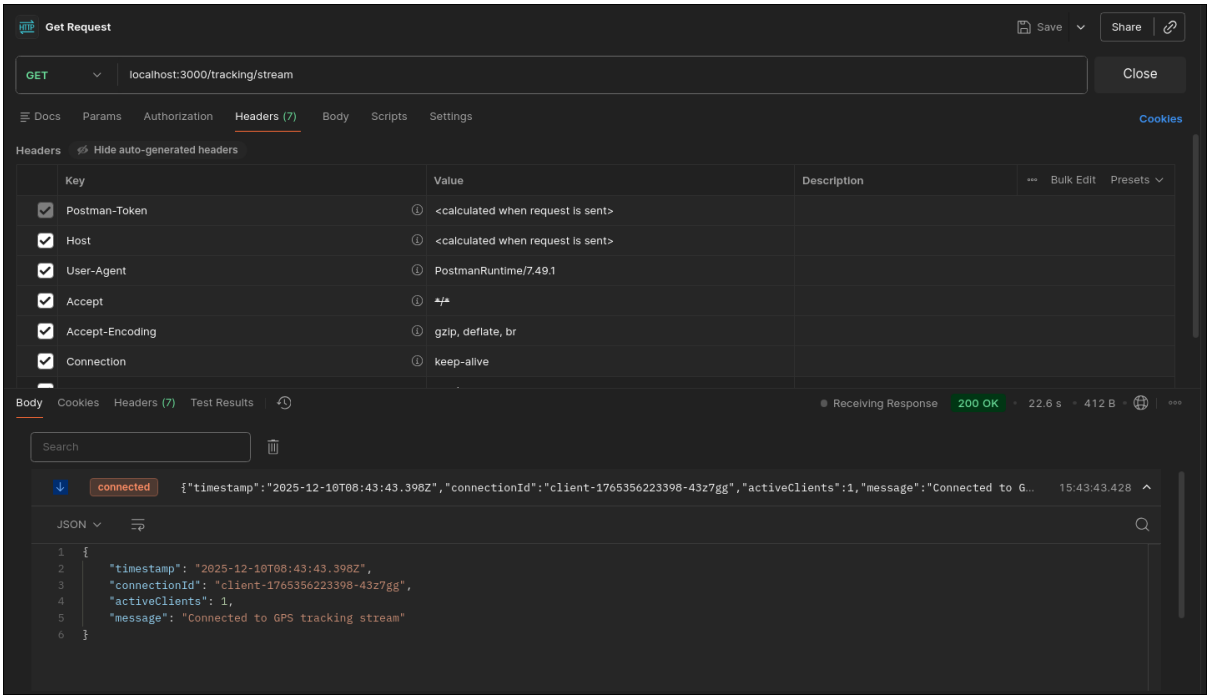
2. Jalankan runner dengan konfigurasi modul auth dan tracking aktif. Pastikan modul load berhasil, migrasi berjalan, topic berhasil di-subscribe, dan schema registry berhasil terkoneksi.

```
$ cat configs/modules.json && bun run dev
{
   "tracking": {
      "enabled": true,
      "packageName": "@ev-connect/tracking"
   },
   "route": {
      "enabled": false,
      "packageName": "@ev-connect/route"
   },
   "auth": {
      "enabled": true,
      "packageName": "@ev-connect/auth"
   }
}
$ bun --watch src/app2.ts
[08:38:50 UTC] INFO: Module added
   module: "tracking"
[08:38:50 UTC] INFO: Module added
   module: "auth"
[08:38:50 UTC] INFO: Resolved module load order
   order: [
     "auth",
```

```
    "tracking"
  ]
[08:38:50 UTC] INFO: Initializing module
  module: "auth"
[08:38:50 UTC] INFO: Running module migrations
  module: "auth"
  folder: "/home/sultan/PA/fms/modules/auth/migrations"
[08:38:50 UTC] INFO: No pending migrations
  module: "auth"
[08:38:50 UTC] INFO: Fastify JWT plugin registered
[08:38:50 UTC] INFO: Module initialized successfully
  module: "auth"
[08:38:50 UTC] INFO: Initialized
  module: "auth"
  ms: 28
[08:38:50 UTC] INFO: Connection manager initialized
  batchSize: 100
  batchInterval: 10
[08:38:50 UTC] INFO: Initializing tracking module
  topic: "vehicle-location"
  schemaRegistry: "http://10.10.11.4:8085"
[08:38:50 UTC] INFO: Initializing Schema Registry connection
  host: "http://10.10.11.4:8085"
[08:38:50 UTC] INFO: Schema Registry connection established
  host: "http://10.10.11.4:8085"
[08:38:50 UTC] INFO: Initializing message processor worker pool
  poolSize: 7
  workerScript:
"/home/sultan/PA/fms/modules/tracking/dist/workers/messageProcessor.js"
  schemaRegistryHost: "http://10.10.11.4:8085"
[08:38:50 UTC] INFO: Using shared worker pool from infrastructure
[08:38:50 UTC] INFO: Message processor ready for processing
[08:38:50 UTC] INFO: Tracking module initialized successfully
[08:38:50 UTC] INFO: Initialized
  module: "tracking"
  ms: 3
[08:38:50 UTC] INFO: Registering module routes and hooks
  module: "auth"
[08:38:50 UTC] INFO: Module routes and hooks registered successfully
  module: "auth"
[08:38:50 UTC] INFO: Registered
  module: "auth"
  ms: 1
```

```
[08:38:50 UTC] INFO: Registering tracking module routes
[08:38:50 UTC] INFO: Tracking module routes registered
[08:38:50 UTC] INFO: Registered
    module: "tracking"
    ms: 0
[08:38:50 UTC] INFO: All modules loaded, initialized, and registered
    count: 2
[08:38:50 UTC] INFO: [KafkaIPC] UDS server listening on /tmp/kafka-worker-
2703247.sock
[08:38:50 UTC] INFO: [KafkaIPC] Starting Kafka worker:
/home/sultan/PA/fms/runner-skeleton/src/kafka-worker.js
[08:38:50 UTC] INFO: Server listening at http://127.0.0.1:3000
[08:38:50 UTC] INFO: Server listening at http://172.16.0.154:3000
[08:38:50 UTC] INFO: Server listening at http://172.16.0.27:3000
[08:38:50 UTC] INFO: Server listening at http://172.22.0.1:3000
[08:38:50 UTC] INFO: └── (empty root node)
    ├── /
    │   ├── health (GET, HEAD)
    │   ├── a
    │   │   ├── dmin/workers/stats (GET, HEAD)
    │   │   └── uth/migrations/status (GET, HEAD)
    │   └── tracking/
    │       ├── st
    │       │   ├── ream (GET, HEAD)
    │       │   └── atus (GET, HEAD)
    │       └── vehicle/
    │           └── :vin (GET, HEAD)
    └── * (OPTIONS)
[08:38:50 UTC] INFO: Server listening
    port: 3000
[08:38:50 UTC] INFO: [KafkaIPC] Worker connected to UDS
[08:38:50 UTC] INFO: [KafkaWorker] Worker connected to UDS
[08:38:50 UTC] INFO: [KafkaWorker] Kafka ready. Subscribed: vehicle-location
```

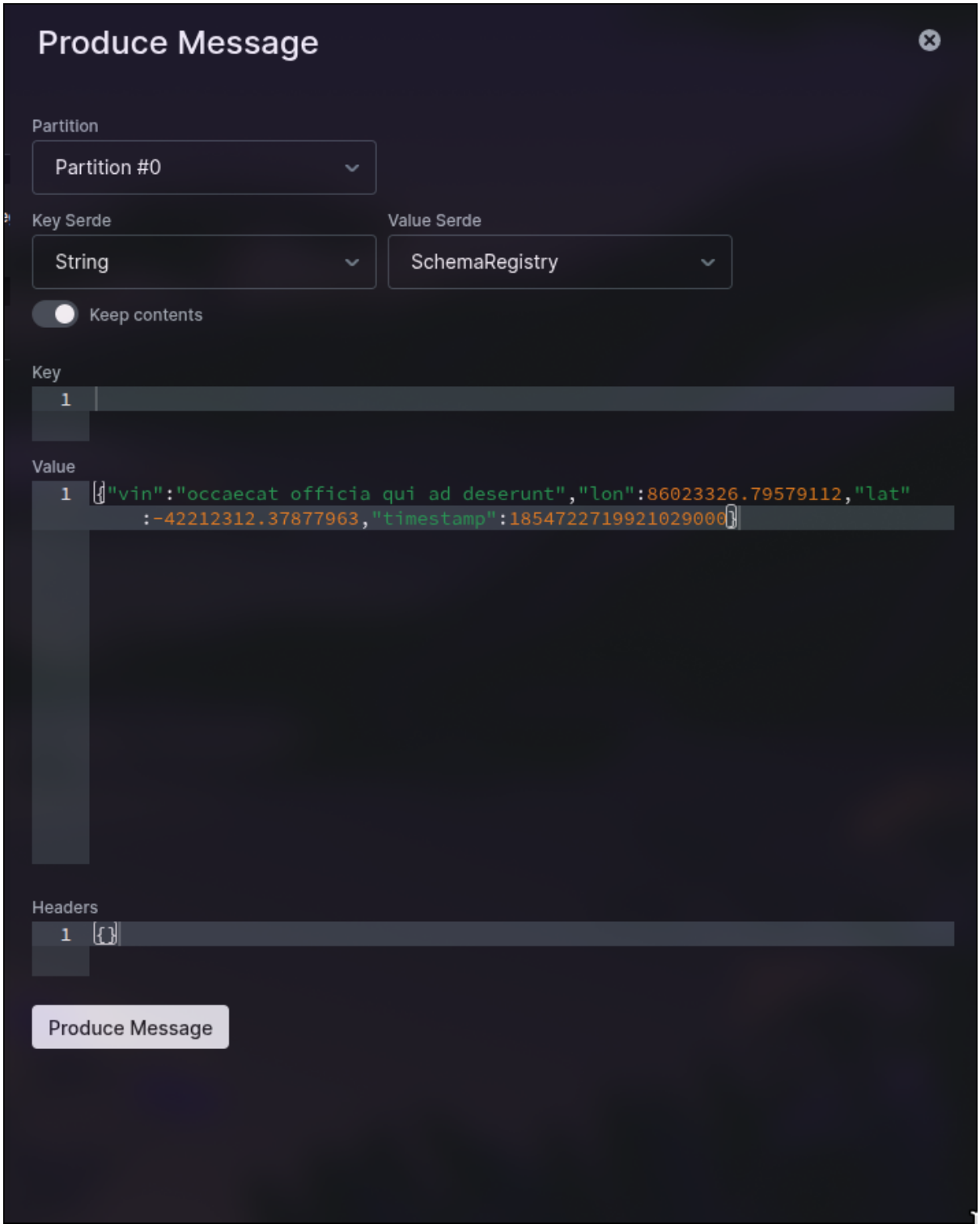
3. Mock koneksi SSE client menggunakan postman kepada endpoint http://localhost:3000/tracking/stream



4. Lakukan pengecekan pada endpoint tracking/status untuk memastikan klien SSE terkoneksi dari sisi server.

```
$ curl -s http://localhost:3000/tracking/status | jq
{
  "status": "active",
  "kafka": {
    "connected": true,
    "topic": "vehicle-location"
  },
  "schemaRegistry": {
    "url": "http://10.10.11.4:8085",
    "status": "connected"
  },
  "sse": {
    "activeClients": 1
  },
  "timestamp": "2025-12-10T08:53:43.384Z"
}
```

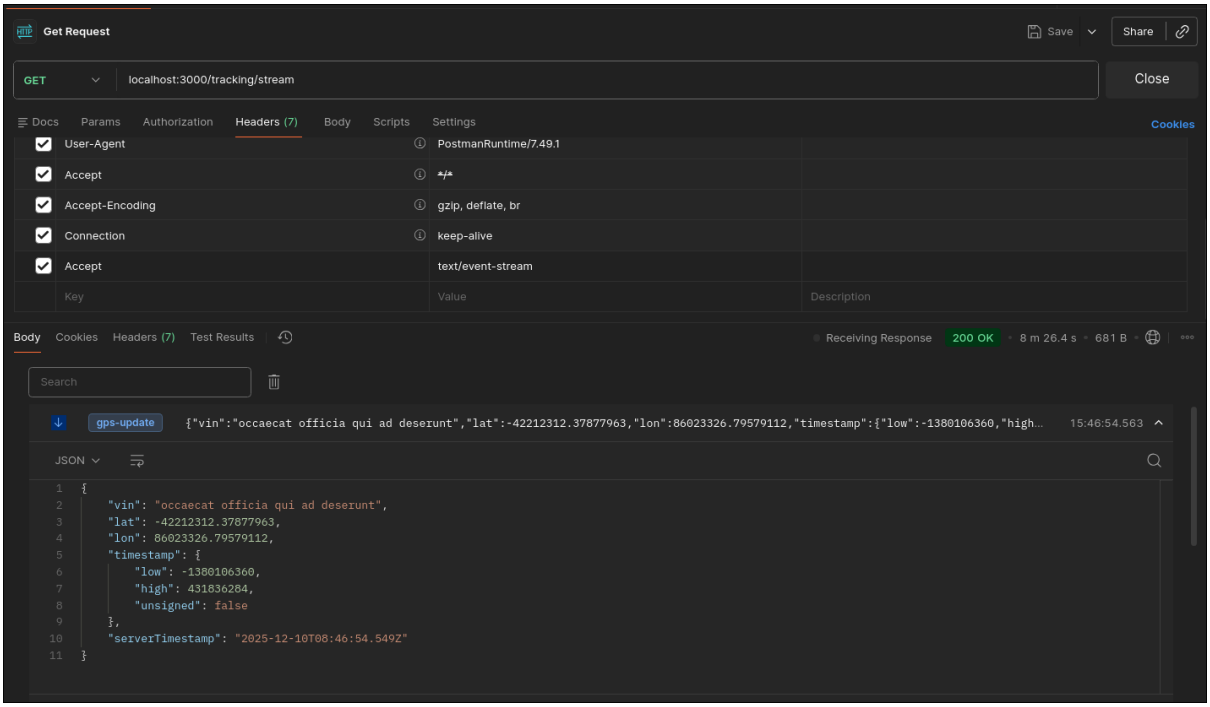
5. Produce kafka message untuk topic vehicle-location secara manual dengan menggunakan Kafbat/kafka-ui. Pastikan message yang di-produce sesuai dengan schema registry



6. Amati log aplikasi untuk indikasi pemrosesan oleh modul tracking (mis. "message received/processed").

```
[08:46:54 UTC] INFO: GPS update received
    vin: "occaecat officia qui ad deserunt"
    lat: -42212312.37877963
    lon: 86023326.79579112
```

7. Pada klien SSE, lakukan pengecekan untuk melihat apakah berhasil menerima event gps-update dengan payload sesuai dengan message yang tadi di-produce.


Get Request
Save
Share
GET
localhost:3000/tracking/stream
Close
Docs
Params
Authorization
Headers (7)
Body
Scripts
Settings
Cookies
User-Agent
PostmanRuntime/7.49.1
Accept
/
Accept-Encoding
gzip, deflate, br
Connection
keep-alive
Accept
text/event-stream
Key
Value
Description
Body
Cookies
Headers (7)
Test Results
Receiving Response
200 OK
8 m 26.4 s
681 B
Search
gps-update
{"vin":"occaecat officia qui ad deserunt","lat":-42212312.37877963,"lon":86023326.79579112,"timestamp":{"low":-1380106360,"high...
15:46:54.563
JSON
{
"vin": "occaecat officia qui ad deserunt",
"lat": -42212312.37877963,
"lon": 86023326.79579112,
"timestamp": {
"low": -1380106360,
"high": 431836284,
"unsigned": false
},
"serverTimestamp": "2025-12-10T08:46:54.549Z"
}

Lampiran 4

Skenario D

1. Clone struktur monorepo

```
$ git clone https://ev-gitlab.mataelang.net/ev-connect/create-fms/backend-module-loader/dev-monorepo
Cloning into 'dev-monorepo'...
warning: redirecting to https://ev-gitlab.mataelang.net/ev-connect/create-fms/backend-module-loader/dev-monorepo.git/
remote: Enumerating objects: 10, done.
remote: Total 10 (delta 0), reused 0 (delta 0), pack-reused 10 (from 1)
Receiving objects: 100% (10/10), 38.95 KiB | 9.74 MiB/s, done.
```

2. Pastikan struktur folder mengandung runner-skeleton/, modules/, serta file package.json

```
$ tree -L 2 -I 'node_modules'
.
├── flow-dev.md
├── flow-prod.md
├── modules
│   └── make-module
├── package.json
├── package-lock.json
├── runner-skeleton
└── temp.md

3 directories, 6 files
```

3. Jalankan bun install pada root direktori monorepo

```
$ bun install
bun install v1.2.12 (32a47ae4)
 node-rdkafka [1/1] warn: node-rdkafka's postinstall cost you 30.9s
warn: node-rdkafka's postinstall cost you 1m1.9s
warn: node-rdkafka's postinstall cost you 1m31.9s

119 packages installed [93.39s]
```

4. Masuk ke direktori modules, clone repositori tracking, install dependencies, dan jalankan build

```
$ cd modules && git clone https://ev-gitlab.mataelang.net/ev-connect/create-fms/backend-modules/tracking
Cloning into 'tracking'...
warning: redirecting to https://ev-gitlab.mataelang.net/ev-connect/create-fms/backend-modules/tracking.git
remote: Enumerating objects: 200, done.
remote: Counting objects: 100% (132/132), done.
remote: Compressing objects: 100% (128/128), done.
remote: Total 200 (delta 53), reused 2 (delta 2), pack-reused 68 (from 1)
Receiving objects: 100% (200/200), 143.89 KiB | 594.00 KiB/s, done.
Resolving deltas: 100% (84/84), done.

$ cd tracking && bun install && bun run build
bun install v1.2.12 (32a47ae4)

+ @fastify/sse@0.4.0
+ @kafkajs/confluent-schema-registry@3.9.0

21 packages installed [1.79s]
$ tsc
```

5. Mencoba menjalankan script interactive make-module untuk membuat boilerplate modul baru.

```
& ./make-module

ℹ 🚀 FMS Module Generator
This script will help you create a new FMS module with all necessary configuration files.

Module name (lowercase, hyphens allowed): coba
Module description (FMS module for coba):
Module version (1.0.0):
Author name (FMS Team):
License (MIT):

ℹ 📦 Dependencies
Add a dependency module? [y/N]: n
Target directory (./coba):
```

ℹ 📋 Summary:
Module: @ev-connect/coba
Description: FMS module for coba
Version: 1.0.0
Author: FMS Team
License: MIT
Target: ./coba

Create this module? [Y/n]: y
ℹ 📁 Creating module directory structure...
ℹ 📄 Creating package.json...
ℹ ⚙️ Creating tsconfig.json...
ℹ 🔧 Creating .npmrc...
ℹ 🚀 Creating .gitlab-ci.yml...
ℹ 📝 Creating source files...
ℹ 🗄️ Creating migration utilities...
ℹ ⚙️ Creating knexfile.ts...
ℹ 📚 Creating README.md...
ℹ 🚫 Creating .gitignore...
ℹ 🔧 Initializing git repository...
Initialized empty Git repository in /home/sultan/dev-monorepo/modules/coba/.git/
[main (root-commit) 3eeab8c] Initial commit: Add coba module scaffold
11 files changed, 765 insertions(+)
create mode 100644 .gitignore
create mode 100644 .gitlab-ci.yml
create mode 100644 .npmrc
create mode 100644 README.md
create mode 100644 knexfile.ts
create mode 100644 package.json
create mode 100644 src/global.d.ts
create mode 100644 src/index.ts
create mode 100644 src/migration.ts
create mode 100644 src/types.ts
create mode 100644 tsconfig.json
✅🎉 Module 'coba' created successfully!

ℹ Next steps:
1. cd ./coba
2. bun install
3. Implement your module logic in src/index.ts

```
  4. Create database migrations in migrations/ directory
  5. bun run build
  6. git push origin main (to trigger publish)

i Migration file naming convention:
  migrations/YYYYMMDDHHMMSS_description.ts
  Example: migrations/20240101120000_create_users_table.ts

i Migration system (Hybrid Approach):
  - All modules share a single 'knex_migrations' table
  - Migrations are namespaced: coba::filename.ts
  - No conflicts between modules
  - Global view of all migrations available

$ cd coba && bun install && bun run build
bun install v1.2.12 (32a47ae4)

3 packages installed [10.00ms]
$ tsc
```

6. Setup environment variable pada runner

```
$ cp .env.example .env
// TODO: implement environment variable based on example
```

7. Jalankan runner dengan konfigurasi modul baru dan modul tracking di-enable

```
$ cat configs/modules.json && bun run dev
{
   "tracking": {
      "enabled": true,
      "packageName": "@ev-connect/tracking"
   },
   "route": {
      "enabled": false,
      "packageName": "@ev-connect/route"
   },
   "auth": {
      "enabled": false,
      "packageName": "@ev-connect/auth"
   },
```

```
  "coba": {
    "enabled": true,
    "packageName": "@ev-connect/coba"
  }
}$ bun --watch src/app2.ts
[09:18:01 UTC] INFO: Module added
  module: "tracking"
[09:18:01 UTC] INFO: Module added
  module: "coba"
[09:18:01 UTC] WARN: Dependency missing
  module: "tracking"
  missing: "auth"
[09:18:01 UTC] INFO: Resolved module load order
  order: [
    "tracking",
    "coba"
  ]
```

Lampiran 5

Skenario E

1. Pull runner-skeleton
2. Jalankan bun install
3. Setup environment variable
4. Setup .npmrc yang mengarah ke package registry @ev-connect

```
$ cat .npmrc
@ev-connect:registry=https://ev-gitlab.mataelang.net/api/v4/groups/{group_id}/-
/packages/npm/
//ev-gitlab.mataelang.net/api/v4/groups/43/-
/packages/npm/:_authToken={auth_token}

// TODO: Inject value from cli
```

5. Jalankan bun install untuk modul:
   a. @ev-connect/tracking
      $ bun install @ev-connect/tracking
      [0.06ms] ".env"
      bun add v1.2.12 (32a47ae4)

      installed @ev-connect/tracking@1.1.0

      [6.00ms] done
   b. @ev-connect/auth
      $ bun install @ev-connect/auth
      [0.14ms] ".env"
      bun add v1.2.12 (32a47ae4)

      installed @ev-connect/auth@1.0.0

      16 packages installed [2.19s]
6. Pastikan modul sudah berada pada node_modules/@ev-connect/…. dan terdefinisi pada package.json

```
$ tree -L 2 -I 'node_modules' node_modules/@ev-connect
node_modules/@ev-connect
├── auth
│   ├── dist
│   ├── package.json
│   ├── README.md
│   └── src
```

```
└── tracking
    ├── dist
    ├── package.json
    ├── README.md
    └── src

7 directories, 4 files

$ cat package.json
{
 "name": "runner-skeleton",
 "module": "index.ts",
 "type": "module",
 "private": true,
 "scripts": {
  "build": "rm -rf dist && tsc",
  "build:clean": "npm run build",
  "dev": "bun --watch src/app2.ts",
  "start": "bun src/app2.ts",
  "typecheck": "bun x tsc --noEmit",
  "watch": "bun x tsc --watch --noEmit",
  "clean": "rm -rf dist"
 },
 "devDependencies": {
  "@types/bun": "latest",
  "@types/node": "^20.11.30",
  "typescript": "^5.5.0"
 },
 "dependencies": {
  "@ev-connect/auth": "^1.0.0",
  "@ev-connect/tracking": "^1.1.0",
  "@fastify/cors": "^11.1.0",
  "dotenv": "^17.2.3",
  "fastify": "^5.6.1",
  "knex": "^3.1.0",
  "node-rdkafka": "^3.5.0",
  "pg": "^8.11.5",
  "pino": "^9.3.2",
  "pino-pretty": "^11.2.2",
  "semver": "^7.7.3"
 },
 "trustedDependencies": [
  "protobufjs"
```

```
  ]
}
```

7. Sesuaikan config modules.json sesuai dengan modul yang sudah di-install melalui package manager

```
$ cat configs/modules.json
{
   "tracking": {
      "enabled": true,
      "packageName": "@ev-connect/tracking"
   },
   "auth": {
      "enabled": true,
      "packageName": "@ev-connect/auth"
   }
}
```

8. Jalankan runner, pastikan log menampilkan module load yang sesuai dan migration semua modul berjalan sampai server LISTENING

```
$ bun --watch src/app2.ts
[09:53:17 UTC] INFO: Module added
   module: "tracking"
[09:53:18 UTC] INFO: Module added
   module: "auth"
[09:53:18 UTC] INFO: Resolved module load order
   order: [
     "auth",
     "tracking"
   ]
[09:53:18 UTC] INFO: Initializing module
   module: "auth"
[09:53:18 UTC] INFO: Running module migrations
   module: "auth"
   folder: "/home/sultan/PA/fms-test/runner-skeleton/node_modules/@ev-
connect/auth/migrations"
[09:53:18 UTC] INFO: No pending migrations
   module: "auth"
[09:53:18 UTC] INFO: Fastify JWT plugin registered
[09:53:18 UTC] INFO: Module initialized successfully
   module: "auth"
```

```
[09:53:18 UTC] INFO: Initialized
    module: "auth"
    ms: 28
[09:53:18 UTC] INFO: Connection manager initialized
    batchSize: 100
    batchInterval: 10
[09:53:18 UTC] INFO: Initializing tracking module
    topic: "vehicle-location"
    schemaRegistry: "http://10.10.11.4:8085"
[09:53:18 UTC] INFO: Initializing Schema Registry connection
    host: "http://10.10.11.4:8085"
[09:53:18 UTC] INFO: Schema Registry connection established
    host: "http://10.10.11.4:8085"
[09:53:18 UTC] INFO: Initializing message processor worker pool
    poolSize: 7
    workerScript: "/home/sultan/PA/fms-test/runner-skeleton/node_modules/@ev-
connect/tracking/dist/workers/messageProcessor.js"
    schemaRegistryHost: "http://10.10.11.4:8085"
[09:53:18 UTC] INFO: Using shared worker pool from infrastructure
[09:53:18 UTC] INFO: Message processor ready for processing
[09:53:18 UTC] INFO: Tracking module initialized successfully
[09:53:18 UTC] INFO: Initialized
    module: "tracking"
    ms: 4
[09:53:18 UTC] INFO: Registering module routes and hooks
    module: "auth"
[09:53:18 UTC] INFO: Module routes and hooks registered successfully
    module: "auth"
[09:53:18 UTC] INFO: Registered
    module: "auth"
    ms: 1
[09:53:18 UTC] INFO: Registering tracking module routes
[09:53:18 UTC] INFO: Tracking module routes registered
[09:53:18 UTC] INFO: Registered
    module: "tracking"
    ms: 1
[09:53:18 UTC] INFO: All modules loaded, initialized, and registered
    count: 2
[09:53:18 UTC] INFO: [KafkaIPC] UDS server listening on /tmp/kafka-worker-
2776290.sock
[09:53:18 UTC] INFO: [KafkaIPC] Starting Kafka worker: /home/sultan/PA/fms-
test/runner-skeleton/src/kafka-worker.js
[09:53:18 UTC] INFO: Server listening at http://127.0.0.1:3000
```

```
[09:53:18 UTC] INFO: Server listening at http://172.16.0.154:3000
[09:53:18 UTC] INFO: Server listening at http://172.16.0.27:3000
[09:53:18 UTC] INFO: Server listening at http://172.22.0.1:3000
[09:53:18 UTC] INFO: └── (empty root node)
    ├── /
    │   ├── health (GET, HEAD)
    │   ├── a
    │   │   ├── dmin/workers/stats (GET, HEAD)
    │   │   └── uth/migrations/status (GET, HEAD)
    │   └── tracking/
    │       ├── st
    │       │   ├── ream (GET, HEAD)
    │       │   └── atus (GET, HEAD)
    │       └── vehicle/
    │           └── :vin (GET, HEAD)
    └── * (OPTIONS)

[09:53:18 UTC] INFO: Server listening
   port: 3000
[09:53:18 UTC] INFO: [KafkaIPC] Worker connected to UDS
[09:53:18 UTC] INFO: [KafkaWorker] Worker connected to UDS
[09:53:18 UTC] INFO: [KafkaWorker] Kafka ready. Subscribed: vehicle-location
```