BAB 3
PELAKSANAAN KERJA MAGANG

3.1 Kedudukan dan Koordinasi

Dalam pelaksanaan kegiatan magang di Kopi Chuseyo Digital, peran yang dijalankan adalah sebagai *Web Developer Intern*. Secara administratif, posisi ini berada di bawah naungan *General Manager* yang memfasilitasi aspek operasional dan integrasi budaya kerja. Namun, struktur pelaporan teknis dirancang khusus mengingat sifat proyek yang strategis bagi transformasi digital perusahaan. Oleh karena itu, seluruh tanggung jawab teknis, pengambilan keputusan arsitektur, dan pengarahan proyek disupervisi secara langsung oleh *Chief Executive Officer* (CEO), menempatkan posisi ini pada area krusial dalam peta organisasi perusahaan.

Tanggung jawab utama dari peran tersebut difokuskan pada perancangan dan pembangunan ulang ekosistem situs profil perusahaan (*Company Profile*) dari dasar (*from scratch*). Inisiatif ini diambil guna menggantikan sistem WordPress lama yang dinilai sudah tidak lagi relevan dengan kebutuhan bisnis saat ini. Lingkup pekerjaan mencakup pengembangan antarmuka situs yang lebih modern dan performa tinggi, serta pembuatan sistem pengelolaan data terpusat. Tujuannya adalah memastikan infrastruktur digital perusahaan memiliki fondasi yang kuat untuk mendukung ekspansi bisnis tanpa terbebani oleh utang teknis atau keterbatasan fitur dari sistem terdahulu.

Mengingat struktur organisasi Kopi Chuseyo tergolong ramping (*lean organization*), alur kerja di dalamnya menjadi sangat cair dan sering melintasi batas fungsi. Koordinasi intens dilakukan bersama *General Manager* untuk menyelaraskan spesifikasi sistem dengan kebutuhan Tim *Marketing Communication*, terutama agar sistem mampu menangani aset visual beresolusi tinggi. Sementara itu, validasi logika bisnis untuk fitur manajemen *event* dibahas langsung dengan CEO. Komunikasi tanpa sekat antara pengembang dan manajemen ini menjadi kunci agar sistem yang dibangun benar-benar presisi dengan target strategis perusahaan. [8]

3.2 Tugas yang Dilakukan

Berdasarkan model tersebut, rangkaian aktivitas pengembangan dipetakan ke dalam lima tahapan sebagai berikut:

1. *Requirements* (Analisis Kebutuhan)
   Identifikasi kebutuhan transformasi bisnis dan pemetaan alur kerja manajemen acara dilakukan melalui diskusi intensif dengan pemangku kepentingan untuk menentukan spesifikasi sistem.

2. *Design* (Perancangan Sistem)
   Penyusunan arsitektur teknis yang mencakup perancangan skema basis data, alur integrasi sistem, serta pembuatan *prototype* antarmuka pengguna.

3. *Implementation* (Konstruksi)
   Realisasi rancangan menjadi produk fungsional melalui penulisan kode program untuk membangun layanan *backend*, antarmuka *frontend*, serta integrasi API secara menyeluruh.

4. *Verification* (Verifikasi)
   Pelaksanaan pengujian fungsional (*Functional Testing*) dan validasi performa sistem untuk memastikan bebas dari *bug* dan sesuai dengan spesifikasi awal sebelum diluncurkan.

5. *Maintenance* (Pemeliharaan)
   Penyusunan dokumentasi teknis dan pemantauan kinerja sistem pasca-implementasi melalui dasbor analitik guna menjamin keberlanjutan operasional jangka panjang.

3.3 Uraian Pelaksanaan Magang

Pelaksanaan kerja magang di Kopi Chuseyo berlangsung selama 12 minggu efektif, terhitung mulai bulan Oktober hingga Desember 2025. Seluruh rangkaian kegiatan dilaksanakan dengan mengacu pada jadwal kerja perusahaan dan target pengembangan sistem yang telah disepakati. Rincian aktivitas mingguan selama periode magang dapat dilihat pada Tabel 3.1.

Tabel 3.1. Ringkasan pekerjaan mingguan berdasarkan rencana kerja magang

Minggu Ke-	Pekerjaan yang dilakukan
1	Proses *onboarding*, pengenalan tim, serta pengumpulan data dan definisi kebutuhan proyek.
2	Perbaikan tampilan website berbasis WordPress, penyusunan peta situs (*sitemap*), dan perancangan rangka (*wireframe*) website.
3	Pengembangan desain antarmuka (*High-Fidelity*), inisiasi kode website dengan Next.js, dan integrasi awal API WordPress.
4	Implementasi komponen antarmuka (*Navigation, Portfolio, Service*), penulisan artikel SEO, serta revisi desain dan wireframe.
5	Pengembangan tampilan responsif (*Responsive Design*), kustomisasi data lanjutan (*Advanced Custom Fields*), dan perancangan arsitektur Admin Panel beserta skema basis data (ERD).
6	Inisiasi repositori dan infrastruktur *backend*, finalisasi ERD modul pengguna, serta implementasi sistem autentikasi dan keamanan (JWT).
7	Pengembangan API untuk modul pendukung (Media, Venue, Organizer), pembaruan skema basis data acara, dan pengembangan desain visual halaman pendukung.
8	Implementasi fitur manajemen Acara (*Event Management*), integrasi *Headless CMS* (Strapi), dan konfigurasi *pipeline* CI/CD untuk otomatisasi *deployment*.
9	Optimalisasi konfigurasi server produksi, perbaikan desain antarmuka utama, dan pengembangan sistem antrean (*Queueing System*) untuk sinkronisasi data.
10	Integrasi menyeluruh antara *frontend* dan *backend*, pengujian fungsionalitas sistem (CRUD), serta peluncuran (*deployment*) sistem ke lingkungan produksi.
	Lanjut pada halaman berikutnya

Tabel 3.1 Ringkasan pekerjaan mingguan berdasarkan rencana kerja magang (Lanjutan)

Minggu Ke-	Pekerjaan yang dilakukan
11	Finalisasi fitur halaman berita (*Posts*), implementasi sistem pengiriman proposal via email, integrasi Google Analytics 4, dan pemeliharaan pasca-peluncuran.
12	Penyusunan dokumentasi teknis, evaluasi akhir sistem, dan penyelesaian laporan kegiatan magang.

3.3.1 Analisis Kebutuhan dan Strategi Migrasi Sistem

Dua minggu pertama magang difokuskan untuk mendalami ekosistem bisnis Kopi Chuseyo dan meninjau infrastruktur teknologi yang tersedia. Mengingat posisi sebagai *Web Developer Intern*, fase ini sangat krusial untuk menjamin solusi yang dibangun memiliki fondasi kokoh dan sejalan dengan transformasi perusahaan menjadi *Digital Agency*.

Melalui serangkaian observasi sistem dan diskusi dengan pemangku kepentingan internal, teridentifikasi beberapa aspek kunci yang menjadi dasar pengembangan sistem baru:

1. Analisis Kesenjangan Sistem Lama (*Gap Analysis*)
   Evaluasi terhadap sistem *website* lama yang berbasis WordPress menunjukkan dua kendala utama. Pertama, terdapat ketidaksesuaian arsitektur informasi dengan model bisnis baru. Platform lama yang didesain untuk blog sulit mengakomodasi struktur data acara yang kompleks (seperti relasi artis dan status acara) tanpa modifikasi berlebihan yang berisiko.

   Kedua, ditemukan adanya ketidakselarasan identitas visual (*visual identity misalignment*). Tema dan kerangka kerja sistem lama membatasi kebebasan tim desain dalam memvisualisasikan citra baru Kopi Chuseyo sebagai agensi digital yang modern dan kreatif. Oleh karena itu, kebutuhan akan kustomisasi tampilan (*custom frontend*) menjadi pendorong utama untuk beralih ke arsitektur *Headless CMS*, bukan karena masalah kinerja server.

2. Identifikasi Kebutuhan Bisnis dan Data
   Berdasarkan arahan CEO, prioritas utama pengembangan adalah modernisasi

ekosistem digital yang berfungsi sebagai portofolio agensi sekaligus katalog acara informatif. Kebutuhan data yang disepakati meliputi:

- Katalog Acara (*Event Catalog*): Sistem harus mampu mengelola atribut spesifik seperti Judul, *Slug*, Tanggal, Poster, *Banner*, Status (Akan Datang/Selesai), serta konten detail artis pengisi acara.
- Portofolio Agensi: Sistem harus memfasilitasi publikasi Layanan (*Services*), Studi Kasus Klien, dan Artikel SEO untuk meningkatkan kredibilitas di mata calon mitra bisnis.

3. Definisi Pengguna dan Alur Kerja (*Workflow*)
   Sistem dirancang untuk digunakan secara eksklusif oleh tim internal (Admin, Manajer, CEO). Alur kerja yang ditetapkan adalah *Direct Publishing*, di mana admin dapat menginput data acara atau artikel dan memilih untuk menyimpannya sebagai draf atau langsung mempublikasikannya. Hal ini bertujuan memangkas birokrasi teknis yang sebelumnya menghambat kecepatan pembaruan konten.

4. Kebutuhan Analitik dan Pemantauan
   Sebagai pengganti data penjualan tiket, manajemen membutuhkan wawasan berbasis data untuk mengukur keberhasilan kampanye. Sistem baru diwajibkan memiliki integrasi dengan *Google Analytics* untuk menampilkan metrik vital seperti jumlah pengunjung harian, sumber trafik, dan tingkat retensi audiens langsung pada dasbor admin.

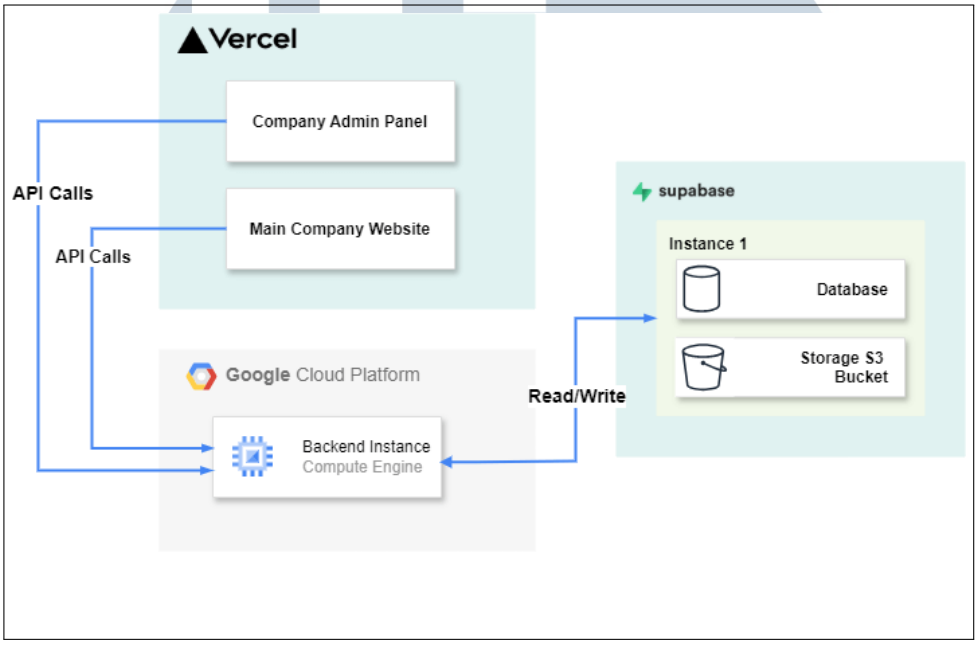
Berangkat dari analisis kebutuhan tersebut, diputuskan untuk menerapkan pendekatan *clean-slate design* dengan membangun sistem baru dari awal. Teknologi yang dipilih adalah Strapi CMS sebagai pusat data dan *NestJS* untuk logika bisnis. Pilihan ini dinilai paling tepat karena menawarkan fleksibilitas struktur data serta kebebasan desain antarmuka yang memang dicari oleh perusahaan [7].

3.3.2 Perancangan Arsitektur Sistem dan Antarmuka

Minggu ketiga difokuskan pada tahap perancangan sistem (*System Design*) sebagai langkah lanjutan dari analisis kebutuhan yang sudah disepakati. Di fase ini, kebutuhan bisnis diterjemahkan menjadi cetak biru (*blueprint*) teknis yang matang. Prioritas utamanya adalah membangun struktur yang modular dan skalabel,

namun tetap efisien dari sisi sumber daya infrastruktur demi menyesuaikan batasan operasional di tahap awal proyek.

A Perancangan Arsitektur Teknologi



Gambar 3.1. Arsitektur Sistem Terkonsolidasi pada Google Cloud Platform dan Supabase

Untuk memisahkan lapisan presentasi (*frontend*) dari manajemen data (*backend*), pola arsitektur *Headless* diadopsi dengan strategi Konsolidasi Sumber Daya (*Resource Consolidation*). Langkah ini diambil agar efisiensi biaya infrastruktur bisa maksimal tanpa harus mengorbankan performa aplikasi.

Sebagaimana diilustrasikan pada Gambar 3.1, arsitektur sistem dibagi menjadi tiga lapisan utama yang saling terhubung:

1. Lapisan Presentasi (*Frontend Layer*)
   Di sisi kiri diagram, terdapat dua aplikasi klien: *Company Admin Panel* dan *Main Company Website*. Keduanya dibangun berbasis *Next.js* dan diletakkan di platform *Vercel*. Lapisan ini sengaja dipisahkan dari server utama demi memanfaatkan fitur *Edge Network* Vercel. Tujuannya adalah menjaga agar akses pengguna akhir tetap cepat dan responsif saat melakukan pemanggilan API (*API Calls*).

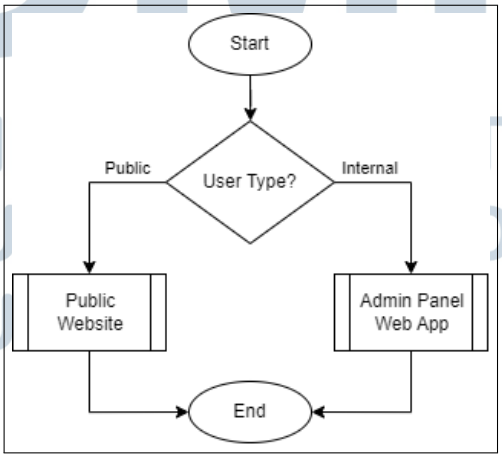
2. Lapisan Komputasi Terpusat (*Consolidated Compute Layer*)
   Bagian tengah diagram menggambarkan infrastruktur di *Google Cloud Platform*. Untuk kebutuhan *backend*, digunakan satu *Compute Engine instance* (tipe `e2-small`) yang menjalankan dua layanan sekaligus:

   - NestJS Service: Berperan sebagai *API Gateway* yang menangani logika bisnis spesifik.
   - Strapi Instance: Berfungsi sebagai sistem manajemen konten berbasis *headless*.

   Kedua layanan ini dijalankan bersamaan dalam satu lingkungan sistem operasi *Ubuntu*. Konfigurasi ini dipilih untuk memangkas biaya operasional secara signifikan, ketimbang harus menggunakan dua server yang terpisah.

3. Lapisan Data Terpadu (*Unified Data Layer*)
   Di sisi kanan diagram, pengelolaan data dipusatkan pada layanan *Supabase* yang berperan sebagai *Single Source of Truth*. Baik NestJS maupun Strapi diarahkan ke satu *instance* basis data *PostgreSQL* yang sama. Sementara itu, penyimpanan aset media dipisahkan ke layanan *Storage S3 Bucket*. Langkah ini krusial untuk menjaga performa basis data agar tidak terbebani oleh penyimpanan fail statis yang besar.

Untuk melengkapi gambaran struktural tersebut, alur kerja sistem juga divisualisasikan dalam bentuk flowchart sistem umum. Visualisasi ini digunakan untuk menunjukkan bagaimana sistem merespons perbedaan jenis pengguna pada tingkat konseptual, tanpa membahas detail teknis implementasi.



Gambar 3.2. Flowchart Sistem Umum

Berdasarkan Gambar 3.2, sistem membedakan alur akses antara pengguna publik dan pengguna internal. Pengguna publik diarahkan untuk mengakses website utama guna memperoleh informasi, sementara pengguna internal mengakses aplikasi admin panel untuk melakukan pengelolaan data. Pemisahan alur ini memungkinkan sistem melayani kebutuhan informasi dan operasional secara terstruktur, namun tetap terintegrasi dalam satu lingkungan backend yang sama.

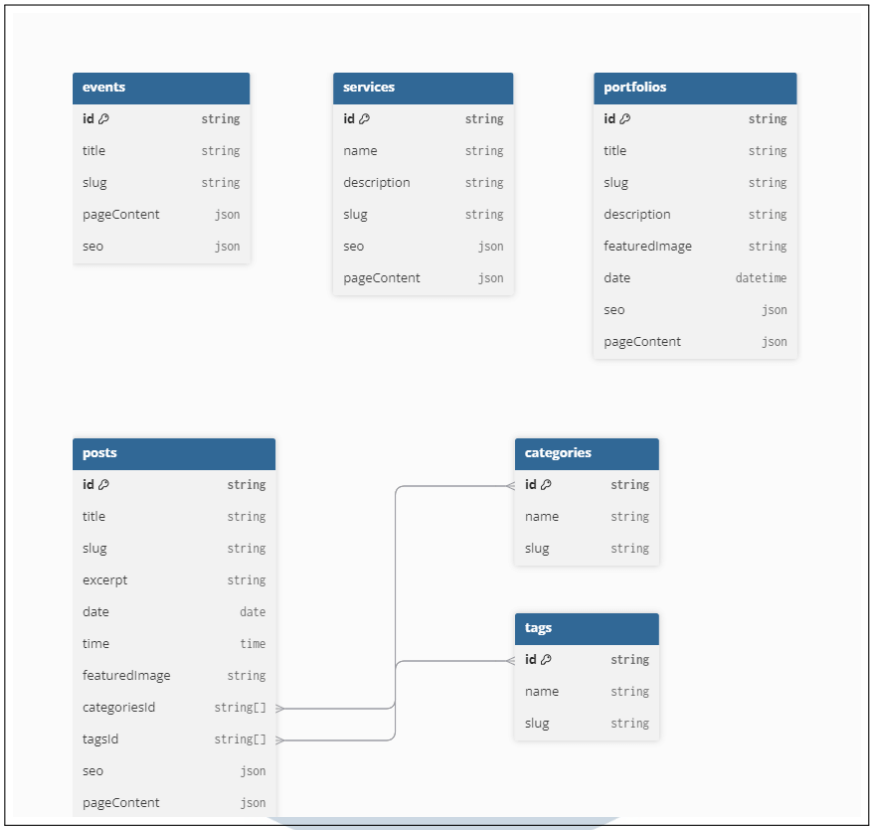
B Perancangan Model Data

Setelah desain infrastruktur terbentuk, fokus bergeser ke perancangan skema basis data (*Database Schema*). Karena sistem ini harus melayani dua kebutuhan berbeda—manajemen konten publik dan operasi bisnis internal—maka diterapkan strategi pemisahan model data (*Data Segregation Strategy*). Dengan pendekatan ini, penyimpanan data dipilah menjadi dua skema logis: skema konten (*Content Schema*) yang berada di ranah Strapi CMS, dan skema aplikasi (*Application Schema*) yang ditangani langsung oleh layanan *backend* NestJS.

B.1 Skema Manajemen Konten (Strapi ERD)

Skema ini dikhususkan untuk menangani data publik yang cenderung statis dan informatif. Fitur *Content-Type Builder* bawaan Strapi dimanfaatkan untuk menyusun struktur data yang fleksibel. Seperti terlihat pada Gambar 3.3, berikut adalah entitas-entitas utamanya:

- Entitas Publikasi (*Posts*): Komponen vital dalam strategi *Content Marketing*. Di dalamnya mencakup atribut `slug` demi URL yang ramah SEO, `excerpt` sebagai ringkasan, serta relasi ke `categories` dan `tags`.
- Entitas Portofolio (*Portfolios*): Berfungsi menampilkan rekam jejak agensi. Entitas ini memuat detail proyek secara lengkap, mulai dari gambar unggulan (`featuredImage`) hingga deskripsi teknis pengerjaannya.
- Entitas Layanan (*Services*): Berisi katalog layanan digital yang ditawarkan. Atribut `pageContent` di sini sengaja dibuat bertipe JSON agar tata letak halaman bisa diatur secara dinamis sesuai kebutuhan.



Gambar 3.3. Rancangan Entity Relationship Diagram untuk Konten pada Strapi CMS

B.2 Skema Manajemen Sistem (*Backend* ERD)

Berbeda dengan skema konten yang berfokus pada penyajian informasi, skema sistem dirancang untuk menangani integritas transaksional, logika bisnis yang kompleks, dan keamanan. Skema ini dikelola menggunakan *Object-Relational Mapping* (ORM) Prisma pada layanan NestJS. Mengacu pada Gambar 3.4, model ini memiliki karakteristik relasional yang kuat antar-entitas:

- Entitas `events` menyimpan seluruh data operasional acara, termasuk judul, deskripsi, serta jadwal waktu pelaksanaan yang direpresentasikan melalui atribut startAt dan endAt.
- Entitas `venues` berfungsi untuk mengelola data lokasi atau tempat pelaksanaan acara secara terpusat, sehingga informasi alamat dan koordinat tetap konsisten dan terhindar dari duplikasi.
- Entitas `organizers` digunakan untuk menyimpan profil lengkap penyelenggara acara yang nantinya akan direlasikan sebagai pemilik

atau penanggung jawab dari setiap entitas acara yang terdaftar.

- Entitas `medias` berperan dalam manajemen aset digital atau berkas multimedia, seperti poster dan *banner*, yang digunakan sebagai komponen visual pendukung informasi acara.
- Entitas `users` menyimpan data kredensial akses dan informasi akun pengguna, yang menjadi fondasi utama dalam penerapan sistem keamanan serta kontrol hak akses pada aplikasi.

Gambar 3.4. Rancangan Entity Relationship Diagram untuk Sistem *Backend*

Pemisahan model data ini memberikan keuntungan ganda: performa tinggi untuk penyajian konten melalui Strapi, dan integritas data yang ketat untuk operasi bisnis melalui struktur relasional *backend* [13].

C Perancangan Antarmuka Pengguna (UI/UX)

Fase perancangan sistem ditutup dengan pematangan aspek pengalaman pengguna (*User Experience*) dan antarmuka visual (*User Interface*). Perangkat lunak *Figma* digunakan untuk menyusun purwarupa tingkat tinggi (*high-fidelity*

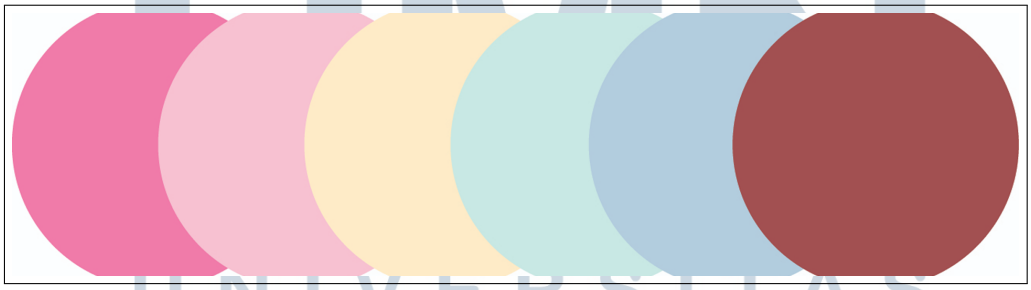
prototype), yang nantinya berfungsi sebagai acuan visual utama bagi proses implementasi *frontend*. Fokus perancangannya terbagi menjadi dua area utama:

C.1 Konsep Visual dan Identitas Agensi

Transformasi Kopi Chuseyo dari bisnis *Food & Beverage* ke agensi digital jelas memerlukan penyegaran identitas visual. Tantangan terbesarnya adalah menggeser persepsi pengguna dari nuansa "kafe yang ceria" menjadi "mitra bisnis profesional", tanpa harus kehilangan karakter unik yang sudah melekat. Oleh sebab itu, perancangan visual difokuskan pada perombakan tiga elemen dasar: Palet Warna, Tipografi, dan Komponen Antarmuka.

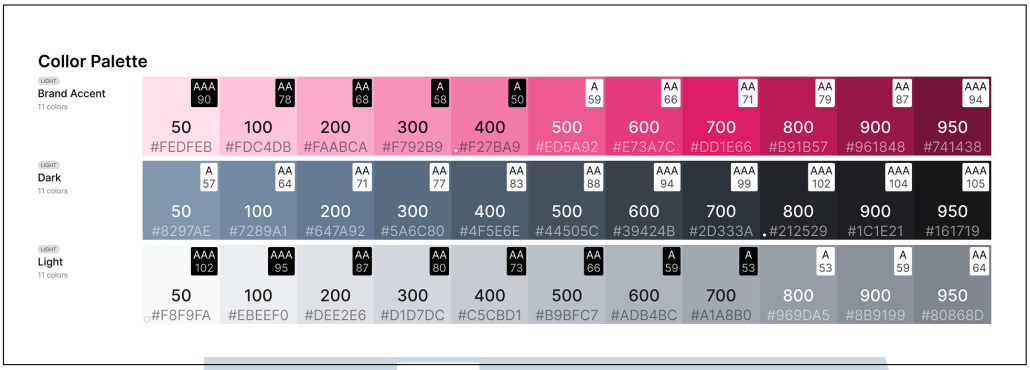
C.1.1 Transformasi Palet Warna (*Color Palette*)

Proses desain dimulai dengan meninjau kembali identitas visual yang sudah ada. Langkah ini penting untuk melihat seberapa jauh perbedaan antara citra lama dengan citra agensi digital yang ingin dibangun. Seperti terlihat pada Gambar 3.5, identitas sebelumnya didominasi oleh warna-warna pastel yang cukup ramai, mulai dari merah muda, kuning, hingga biru. Walaupun warna-warna ini sukses membangun suasana kafe yang ceria, nuansanya terasa terlalu santai untuk pasar bisnis atau B2B. Klien perusahaan biasanya lebih mencari mitra yang terlihat terpercaya dan profesional, bukan sekadar terlihat ramah.



Gambar 3.5. Skema Warna Lama: Dominasi Warna Pastel Khas Kafe

Untuk menjawab kebutuhan tersebut, sistem warna diubah menjadi lebih dewasa. Seperti yang disajikan pada Gambar 3.6, desain kini menggunakan skema Monokromatis sebagai fondasi utama. Perubahan ini bertujuan agar tampilan visual menjadi lebih tegas, bersih, dan tidak lagi terjebak pada nuansa kafe seperti sebelumnya.



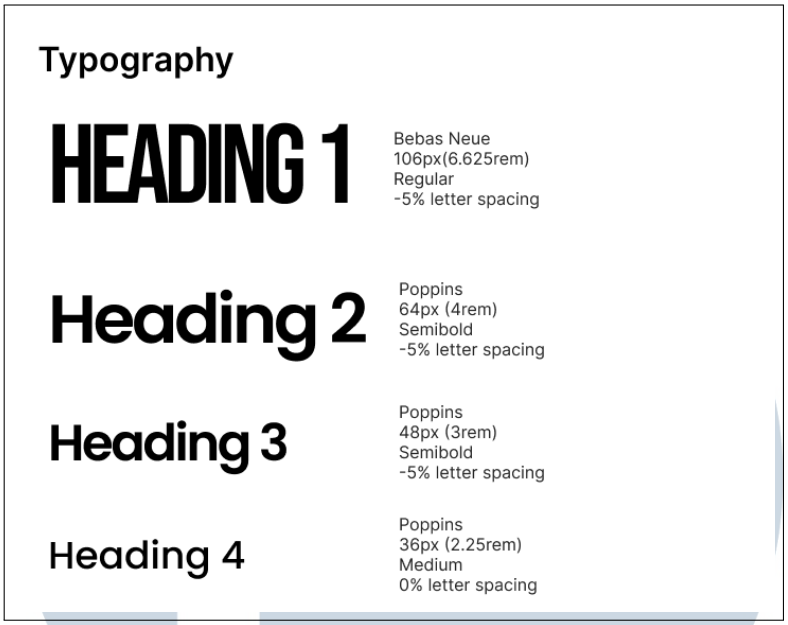
Gambar 3.6. Sistem Palet Warna Baru: Transisi ke Skema Monokrom dengan Aksen Identitas

Komposisi warna ini dirancang dengan mempertimbangkan beberapa aspek kunci sebagai berikut:

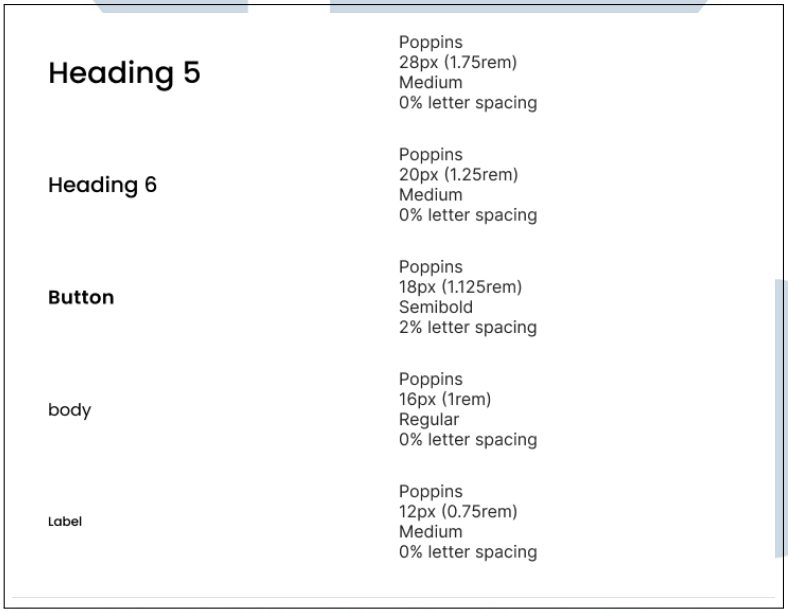
- Dominasi Warna Netral (Penerapan Aturan 60-30): Diterapkan prinsip *The 60-30-10 Rule*, di mana 60% area antarmuka didominasi warna latar terang (putih/abu-abu) dan 30% menggunakan warna sekunder gelap (hitam/abu-abu tua) untuk teks dan struktur. Dalam psikologi warna B2B, penggunaan warna netral ini dipilih untuk membangun persepsi profesionalisme, stabilitas, dan otoritas, sekaligus menciptakan ruang negatif (*whitespace*) yang memadai agar konten portofolio menjadi fokus utama.
- Aksen Identitas (Penerapan *Von Restorff Effect*): Sisa 10% komposisi warna didedikasikan untuk warna *Pink* (skala 50-950) sebagai aksen. Strategi ini memanfaatkan prinsip *Von Restorff Effect* (Efek Isolasi), yang menyatakan bahwa objek yang berbeda secara visual akan lebih menarik perhatian. Dengan membatasi warna pink hanya pada elemen interaktif (seperti tombol *Call-to-Action*), elemen tersebut akan terlihat kontras di atas latar monokrom, secara efektif mengarahkan fokus pengguna untuk melakukan aksi penting.

C.1.2 Pengembangan Tipografi (*Typography System*)

Elemen tipografi dirancang untuk menjembatani dua peran perusahaan sekaligus, yakni sebagai promotor acara yang dinamis dan agensi digital yang profesional. Strategi *Font Pairing* diterapkan dengan cara memadukan dua jenis huruf yang memiliki karakter kontras namun tetap harmonis dan saling mengisi.



(a) Hierarki Judul Utama (Headings)



(b) Detail Elemen Antarmuka dan Bodi Teks

Gambar 3.7. Sistem Tipografi: Kombinasi Bebas Neue dan Poppins dengan Skala Hierarki

Berdasarkan Gambar 3.7, penerapan tipografi dibagi menjadi dua fungsi utama:

- Display Font (Bebas Neue): Digunakan khusus untuk *Heading 1*, font *Bebas Neue* dengan gaya *Condensed Sans Serif* memberikan kesan poster yang tegas dan tinggi. Sebagaimana terlihat pada Gambar 3.7a, ukuran huruf

ditetapkan sebesar 106px dengan *letter spacing* -5% untuk menciptakan dampak visual yang kuat pada bagian *Hero Section*.

- Body & UI Font (Poppins): Untuk judul sekunder (*Heading 2-6*), teks paragraf, dan elemen antarmuka, digunakan jenis huruf *Poppins*. Huruf bergaya geometris ini dipilih karena memiliki tingkat keterbacaan (*legibility*) yang tinggi, sehingga nyaman dilihat di berbagai ukuran. Khusus pada *Heading 2*, diterapkan ketebalan *Semibold* dengan ukuran 64px untuk menjaga kontras dan hierarki visual yang jelas terhadap judul utama.

Sistem ini juga dioptimalkan demi mendukung kenyamanan penggunaan (*Usability*). Sebagaimana diilustrasikan pada Gambar 3.7b, penyesuaian ketebalan huruf dilakukan pada komponen mikro untuk memastikan informasi tersampaikan dengan jelas.

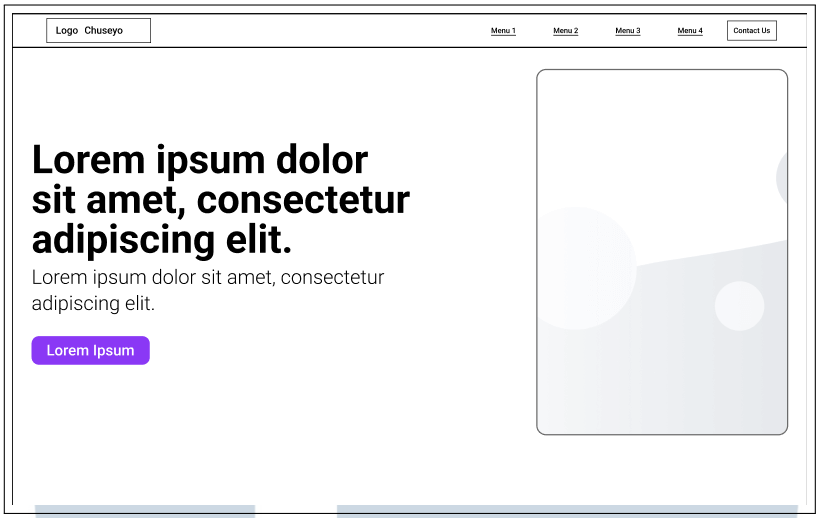
- Tombol (*Button*): Menggunakan ukuran 18px dengan berat *Semibold* dan *letter spacing* 2% untuk memastikan teks aksi terlihat jelas dan mudah diklik.
- Label: Menggunakan ukuran 12px dengan berat *Medium* (bukan *Light*) untuk mencegah teks terlihat kabur pada layar resolusi rendah.

C.2 Perancangan Wireframe Halaman Utama (*Home Page Wireframe*)

Sebelum melangkah ke tahap desain visual yang lebih mendetail, rancangan kerangka kerja (*wireframe*) untuk halaman utama disusun terlebih dahulu. Struktur halaman ini dikembangkan menggunakan pendekatan *Narrative Design*, di mana informasi disajikan mengalir layaknya sebuah cerita. Hierarki informasi kemudian ditata berdasarkan Model AIDA (*Attention, Interest, Desire, Action*) untuk memandu alur pemahaman pengunjung secara bertahap, mulai dari sekadar mengetahui hingga tertarik untuk bekerja sama. Oleh karena itu, setiap bagian (*section*) dirancang secara khusus untuk menjawab pertanyaan yang muncul di benak pengguna (*User's Mental Model*) ketika menelusuri halaman.

Berikut adalah rincian elemen struktural berdasarkan alur narasi tersebut:

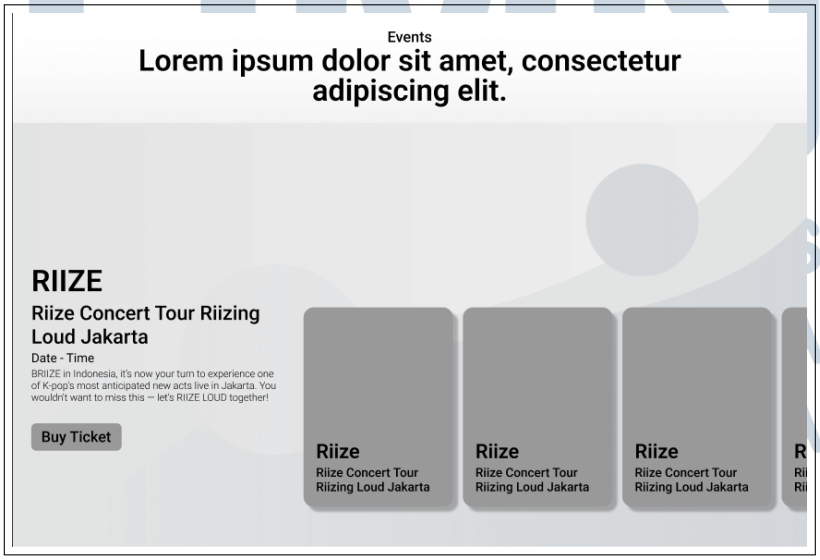
1. Tahap Perhatian (*Attention*) - Hero Section



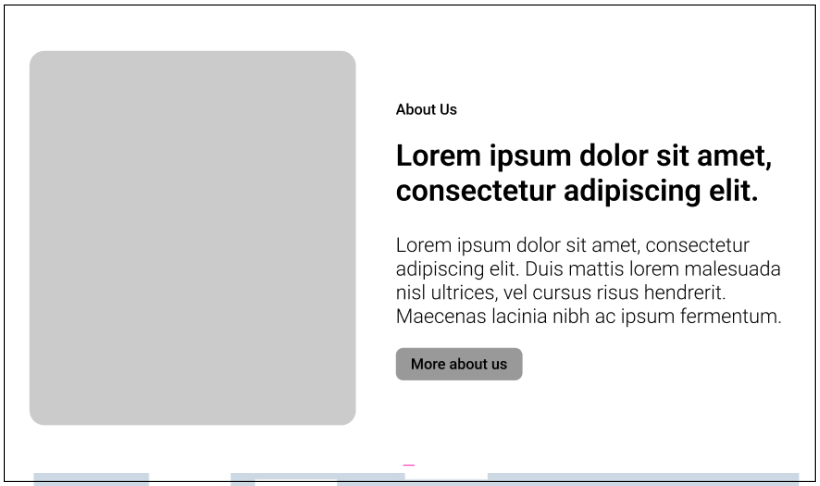
Gambar 3.8. Wireframe Hero Section: Menarik Perhatian dengan Headline Kuat

Bagian ini berfungsi untuk menjawab pertanyaan mendasar pengguna mengenai identitas serta penawaran utama perusahaan. Sebagaimana diilustrasikan pada Gambar 3.8, rancangan *wireframe* menonjolkan penggunaan tipografi berukuran besar jenis *Bebas Neue* sebagai *headline* utama. Elemen ini dipadukan dengan tombol aksi (*CTA*) berdesain kontras guna menarik perhatian visual secara instan. Selain itu, tata letak terlihat diatur agar tetap bersih dengan dominasi ruang kosong atau *negative space*, memastikan pesan utama dapat tersampaikan tanpa gangguan visual.

2. Tahap Ketertarikan (*Interest*) - Events & About



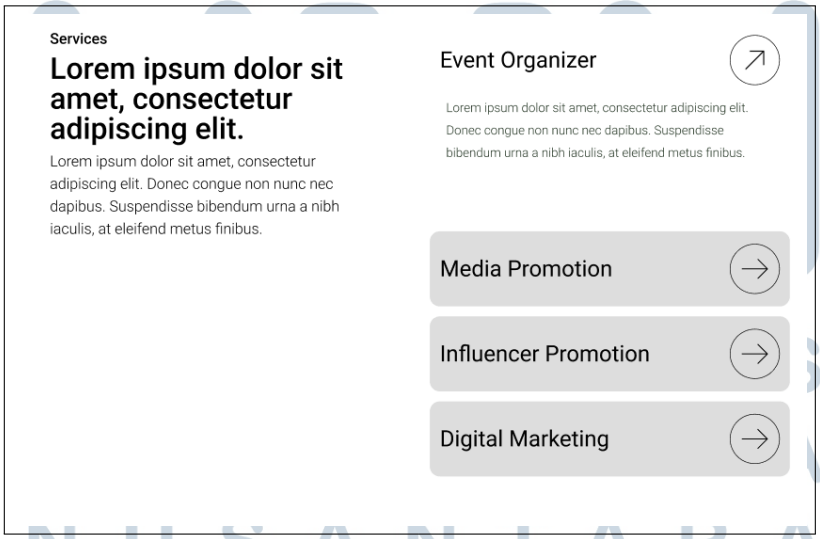
Gambar 3.9. Wireframe Event Section: Tata Letak Katalog Aktivitas



Gambar 3.10. Wireframe About Section: Penjelasan Singkat Identitas Perusahaan

Untuk menjawab keraguan *"Apakah perusahaan ini aktif?"* dan *"Apa kredibilitasnya?"*, struktur halaman menampilkan katalog acara terkini tepat di bawah Hero. Penggunaan kartu acara (*card layout*) pada Gambar 3.9 dengan ruang untuk poster artis memberikan bukti aktivitas yang relevan. Dilanjutkan dengan bagian *About* pada Gambar 3.10 yang menyediakan ruang untuk penjelasan identitas perusahaan secara ringkas.

3. Tahap Informasi Layanan - Service Section

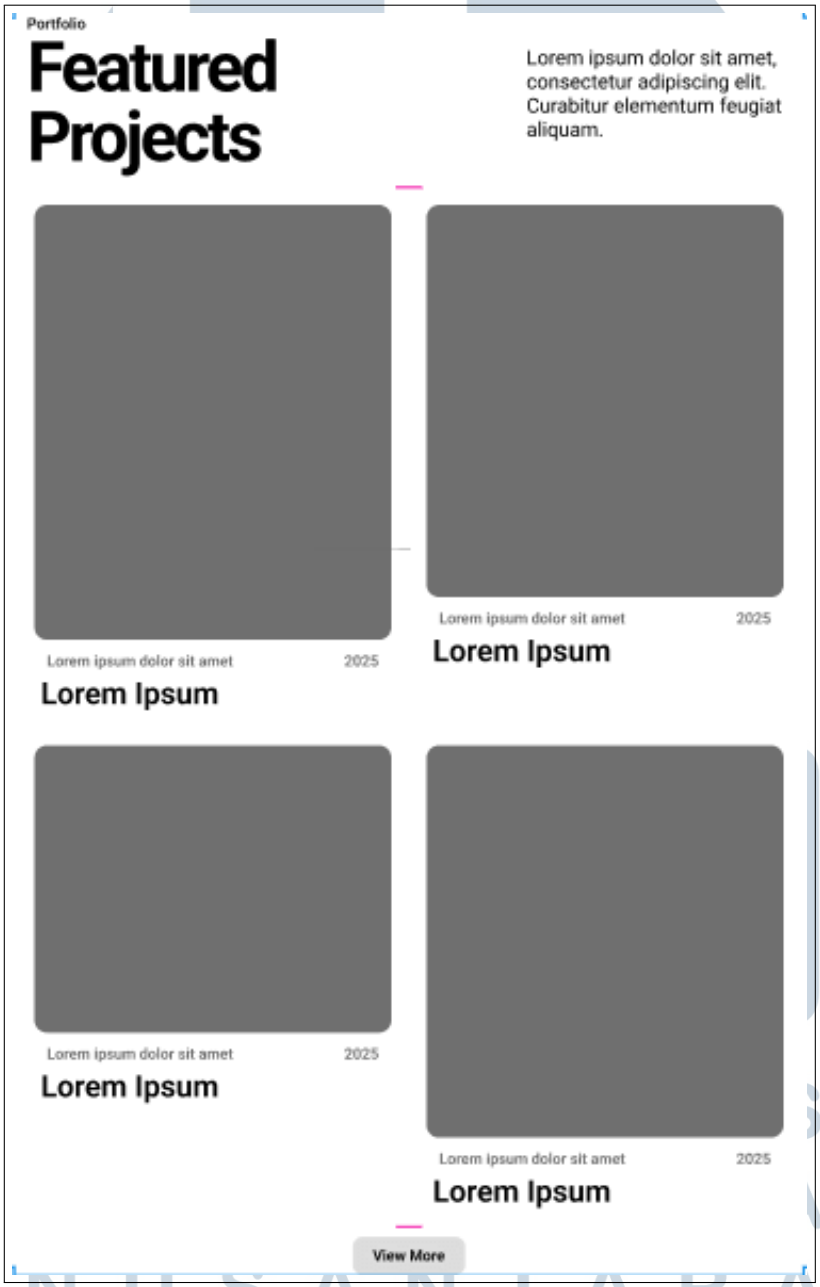


Gambar 3.11. Wireframe Service Section: Katalog Layanan Digital Agency

Menjawab pertanyaan *"Apa saja yang bisa Anda kerjakan?"*, bagian ini menjabarkan lingkup layanan agensi sebagaimana diilustrasikan pada

Gambar 3.11. Desain *wireframe* tersebut menggunakan format daftar (*list*) vertikal dengan indikator ikon panah interaktif, yang mengundang pengguna untuk mengeksplorasi detail setiap layanan lebih lanjut.

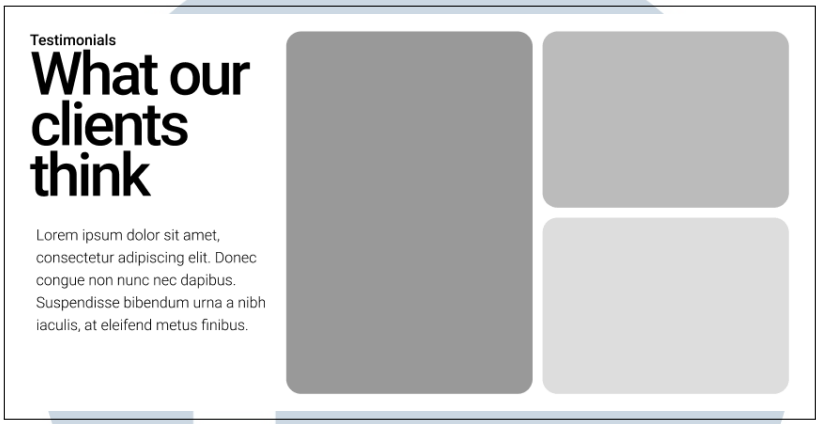
4. Tahap Keinginan (*Desire*) - Portfolio & Testimonial



Gambar 3.12. Wireframe Portfolio Section: Tata Letak Grid Hasil Pengerjaan

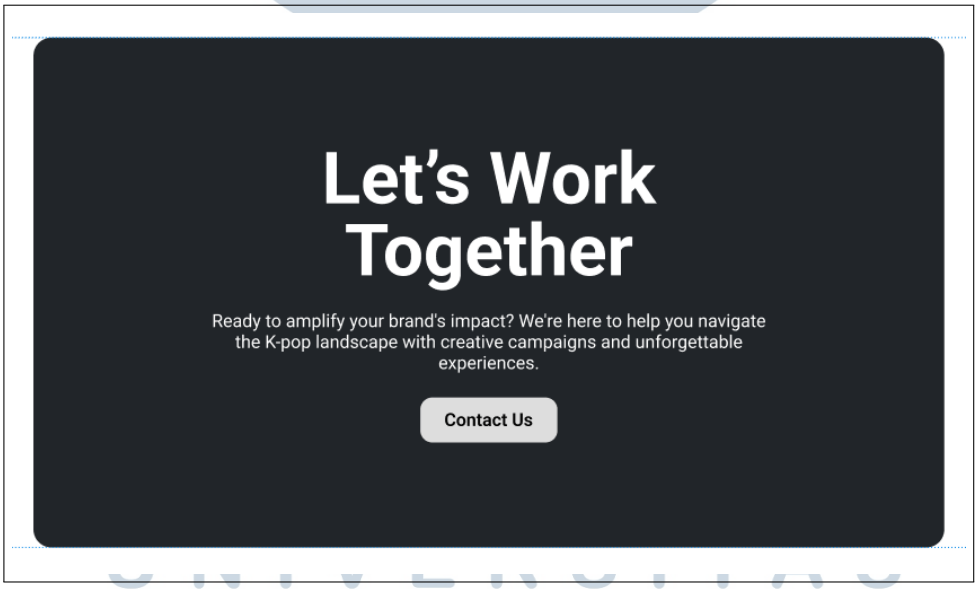
Pada tahap ini, desain berfokus pada *Social Proof*. Bagian Portofolio pada Gambar 3.12 dirancang menggunakan *grid layout* untuk menampilkan

cuplikan proyek yang telah selesai. Didukung dengan bagian Testimonial pada Gambar 3.13 yang menyediakan blok khusus untuk memuat ulasan klien guna memvalidasi kepercayaan pengguna.



Gambar 3.13. Wireframe Testimonial Section: Validasi Kualitas Melalui Ulasan

5. Tahap Aksi (*Action*) - Call To Action (CTA)



Gambar 3.14. Wireframe CTA Section: Area Fokus untuk Konversi Pengunjung

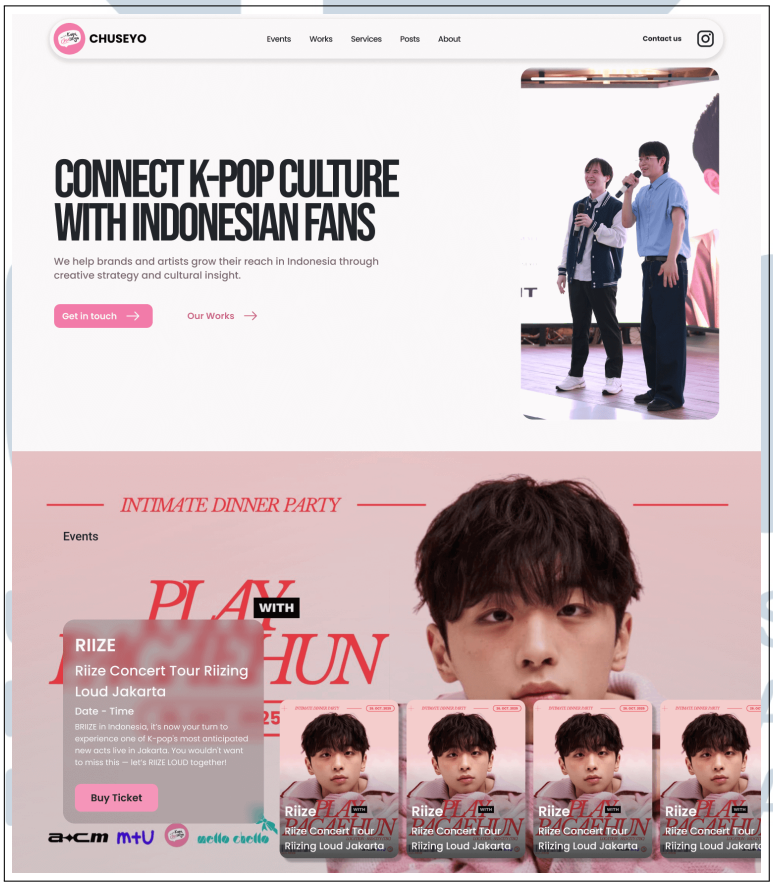
Sebagai penutup alur narasi, bagian ini difungsikan untuk memberikan jawaban mengenai cara memulai kerja sama. Sebagaimana diilustrasikan pada Gambar 3.14, area tersebut dirancang dengan kontras visual yang tinggi menggunakan latar belakang gelap demi menonjolkan teks *"Let's Work Together"* serta tombol aksi utama. Strategi ini bertujuan untuk memotivasi

pengunjung agar segera bertindak dan beralih status dari sekadar pembaca menjadi calon mitra bisnis.

3.3.3 Pengembangan Desain Antarmuka Tingkat Tinggi (*High Fidelity Design*)

Tahap akhir perancangan difokuskan pada pengembangan kerangka *wireframe* menjadi tampilan visual akhir yang utuh atau sering disebut sebagai *High-Fidelity*. Pada fase ini, sistem desain dan prinsip psikologi visual diterapkan untuk menciptakan pengalaman pengguna yang optimal. Selanjutnya, analisis mengenai implementasi desain tersebut akan diuraikan secara mendetail mengikuti urutan alur narasi pada halaman utama.

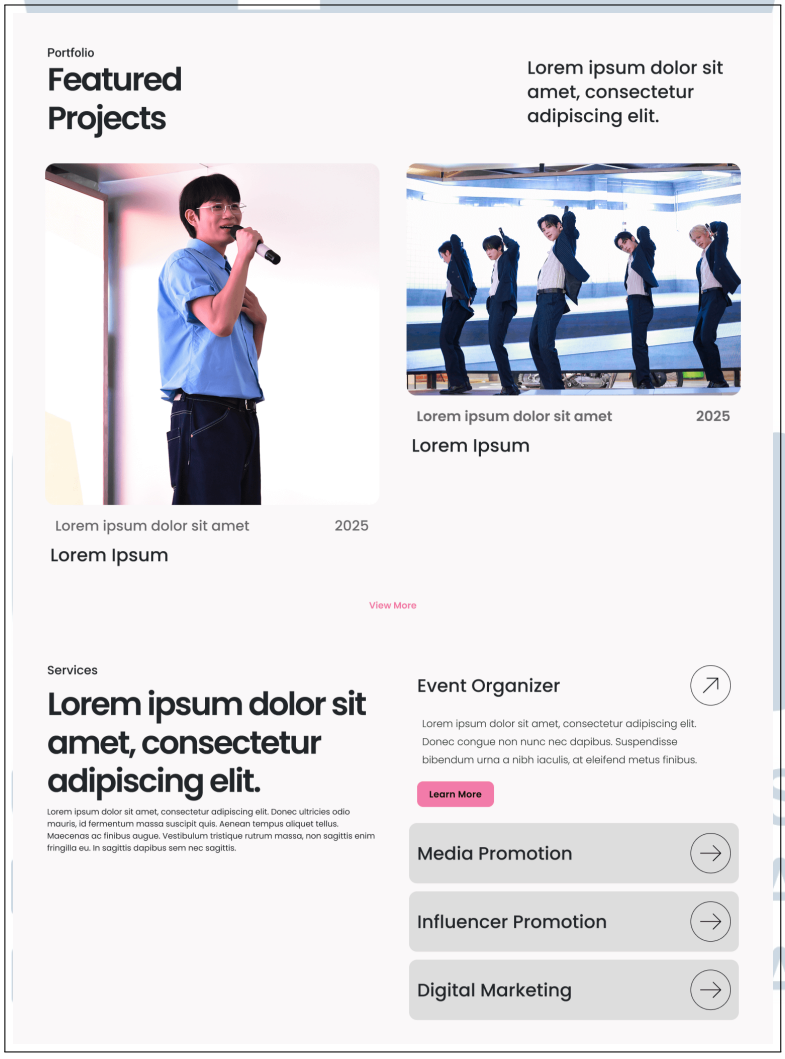
A Segmen Pembuka (Visual Hierarchy)



Gambar 3.15. Segmen Pembuka: Hero Section dan Katalog Acara

Merujuk pada Gambar 3.15, prinsip Hierarki Visual diterapkan melalui pengaturan skala yang cukup kontras. Judul utama ditampilkan dengan ukuran besar 106px, sedangkan teks penjelas dibuat jauh lebih kecil pada ukuran 16px. Perbedaan ukuran yang signifikan ini sengaja dirancang untuk memandu mata pengguna agar menangkap pesan inti terlebih dahulu sebelum membaca detail lainnya. Di sisi lain, penggunaan aset visual berupa foto dokumentasi asli serta poster artis bertujuan untuk memperkuat kredibilitas dan memberikan bukti nyata atas pengalaman agensi.

B Segmen Penawaran (Gestalt & Whitespace)



Gambar 3.16. Segmen Penawaran: Portofolio Proyek dan Daftar Layanan

Pada bagian tengah halaman seperti terlihat di Gambar 3.16, pemanfaatan Ruang Kosong (*Whitespace*) dilakukan secara maksimal di sekitar susunan portofolio dan daftar layanan. Pendekatan ini bertujuan memberikan ”ruang napas” pada desain sehingga tercipta kesan yang lebih elegan dan tidak terasa padat. Selain itu, diterapkan pula prinsip Gestalt (*Proximity*) atau kedekatan jarak dengan menempatkan ikon panah berdampingan erat dengan label teks. Pengaturan posisi ini berfungsi sebagai penanda visual bagi pengguna bahwa kedua elemen tersebut merupakan satu kesatuan tombol interaktif yang utuh.

C Segmen Penutup (Von Restorff Effect)

Gambar 3.17. Segmen Penutup: Bukti Sosial dan Ajakan Bertindak (CTA)

Sebagai penutup alur cerita pada Gambar 3.17, desain beralih ke latar belakang gelap untuk menciptakan kontras visual yang kuat. Tombol aksi (*Call-to-Action*) menggunakan warna aksen Pink yang mencolok di tengah dominasi warna monokrom. Strategi ini memanfaatkan Efek Isolasi (*Von Restorff Effect*), yang secara efektif menarik perhatian pengguna untuk melakukan konversi akhir (menghubungi perusahaan) setelah melihat bukti sosial pada bagian testimoni.

3.3.4 Implementasi *Backend* dan Manajemen Konten

Memasuki minggu kelima, fokus kegiatan beralih pada tahap Implementasi atau *System Construction*. Sesuai dengan arsitektur *Headless* yang digunakan, proses pengembangan diawali dari sisi *backend*. Pendekatan ini dipilih untuk memastikan bahwa struktur data dan layanan API telah tersedia sepenuhnya sebelum pengerjaan antarmuka pengguna dimulai. Sebagai langkah awal, dilakukan instalasi dan konfigurasi Strapi CMS yang berfungsi sebagai pusat manajemen konten. Sistem ini dirancang agar mampu menyajikan data secara fleksibel ke berbagai platform klien tanpa batasan teknis yang kaku.

A Konfigurasi Koneksi Basis Data

Sebagai langkah inisiasi, dilakukan pengaturan pada adaptor basis data untuk menghubungkan aplikasi dengan layanan *Supabase PostgreSQL*. Dalam proses ini, standar keamanan *The Twelve-Factor App* diterapkan guna menjaga kerahasiaan data dengan cara memisahkan konfigurasi dari kode program utama. Dengan demikian, informasi sensitif tidak ditulis secara langsung atau *hardcoded* di dalam skrip, melainkan dibaca melalui mekanisme variabel lingkungan (*Environment Variables*). Implementasi konfigurasi adaptor basis data pada berkas `config/database.ts` dapat dilihat pada Kode 3.1. Pendekatan ini memitigasi risiko kebocoran data sekaligus menjamin portabilitas aplikasi antar-lingkungan (*Development, Staging, Production*).

```
1 import path from "path";
2
3 export default ({ env }) => {
4   const client = env("DATABASE_CLIENT", "postgres");
5
6   const connections = {
```

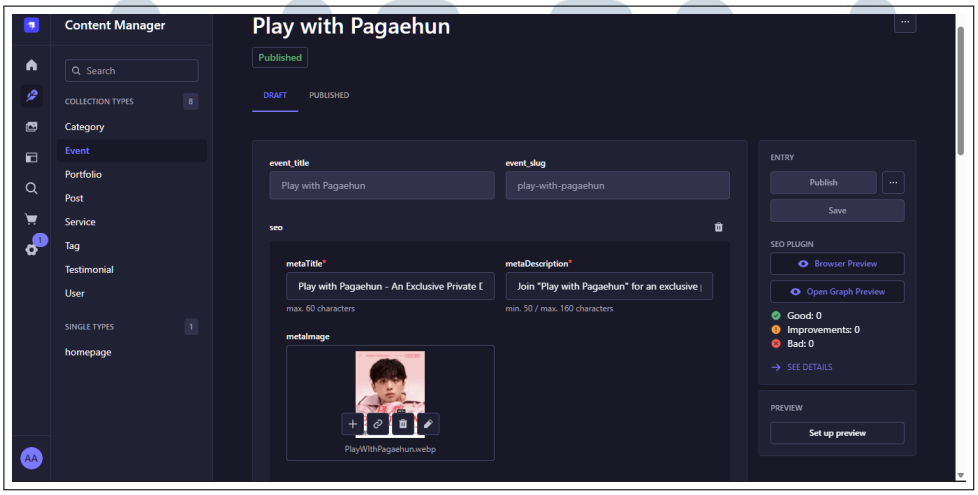
```
    postgres: {
      connection: {
        host: env("DATABASE_HOST", "localhost"),
        port: env.int("DATABASE_PORT", 5432),
        database: env("DATABASE_NAME", "postgres"),
        user: env("DATABASE_USERNAME", "postgres"),
        password: env("DATABASE_PASSWORD"),
        ssl: {
          rejectUnauthorized: env.bool(
            "DATABASE_SSL_REJECT_UNAUTHORIZED",
            false
          ),
        },
        schema: env("DATABASE_SCHEMA", "public"),
        keepAlive: true,
        keepAliveInitialDelayMillis: 10000,
      },
      pool: {
        min: env.int("DATABASE_POOL_MIN", 0),
        max: env.int("DATABASE_POOL_MAX", 6),
        idleTimeoutMillis: env.int("
    DATABASE_IDLE_TIMEOUT", 30000),
        reapIntervalMillis: 1000,
        acquireTimeoutMillis: env.int("
    DATABASE_ACQUIRE_TIMEOUT", 60000),
        createTimeoutMillis: 30000,
        destroyTimeoutMillis: 5000,
        createRetryIntervalMillis: 200,
        propagateCreateError: false,
      },
      afterCreate: (conn, done) => {
        conn.on("error", (err) => {
          console.error("[Database] Connection error:",
     err.message);
        });
        done();
```

```
      },
    },

  return {
    connection: {
      client,
      ...connections[client],
      acquireConnectionTimeout: env.int("
DATABASE_CONNECTION_TIMEOUT", 60000),
    },
  };
};
```

Kode 3.1: Konfigurasi Adaptor PostgreSQL pada Strapi

Mengacu pada Kode 3.1, konfigurasi tersebut menggunakan protokol SSL/TLS (melalui parameter `ssl`) untuk mengenkripsi lalu lintas data antara server aplikasi dan basis data, menjamin integritas dan kerahasiaan data saat transit.

B Antarmuka Manajemen Konten



Gambar 3.18. Tampilan Content Manager Strapi untuk Pengelolaan Data Acara

Implementasi skema data tersebut selanjutnya dikelola melalui antarmuka visual Panel Admin Strapi. Tidak seperti CMS konvensional yang menggabungkan sisi konten dan tampilan dalam satu wadah, antarmuka ini secara khusus difokuskan

hanya pada manajemen struktur data. Selain itu, tahap pengisian data awal atau *data seeding* juga telah dilakukan guna memverifikasi apakah relasi antar-entitas sudah berjalan sesuai rencana. Tampilan antarmuka manajemen data tersebut dapat dilihat secara lengkap pada Gambar 3.18.

C Pengujian Endpoint API (*API Testing*)

Tahap akhir implementasi *backend* adalah validasi interoperabilitas sistem melalui pengujian API. Pengujian ini bertujuan untuk memastikan bahwa format data yang dihasilkan mematuhi standar JSON (*JavaScript Object Notation*) yang dapat diproses oleh berbagai platform klien.

Pengujian integrasi dilakukan dengan mengirimkan permintaan HTTP menggunakan metode `GET` ke alamat `/api/events`. Berdasarkan respons sistem yang tersaji pada Kode 3.2, terlihat adanya penerapan pola *Dynamic Zones*. Fitur ini memungkinkan satu titik akses atau *endpoint* untuk mengirimkan berbagai variasi format komponen secara dinamis dan fleksibel tanpa terpaku pada satu struktur data yang kaku.


```
{
    "data": [
        {
            "id": 97,
            "documentId": "nwhcmzzqhz3yy0qg29d4t9v5",
            "event_title": "Play with Pagaehun",
            "event_slug": "play-with-pagaehun",
            "backend_id": "cmishrrsm0001ffoj99zgssl6",
            "event_content": [
                {
                    "id": 24,
                    "__component": "blog-components.
    rich-text"
                },
                {
                    "id": 9,
                    "__component": "image-gallery.image
    -gallery"
```

```
                },
                {
                    "id": 19,
                    "__component": "two-columns.image-paragraph"
                }
            ],
            "seo": {
                "id": 21,
                "metaTitle": "Play with Pagaehun - An Exclusive Private Dinner",
                "metaDescription": "Join \"Play with Pagaehun\" for an exclusive private dinner..."
            }
        }
    ],
    "meta": {
        "pagination": {
            "page": 1,
            "pageSize": 25,
            "pageCount": 1,
            "total": 1
        }
    }
}
```


Kode 3.2: Ringkasan Respons JSON Endpoint Events

Analisis respons JSON pada Kode 3.2 menunjukkan tiga karakteristik teknis utama:

1. Integrasi Lintas-Sistem
   Keberadaan atribut `backend_id` berfungsi sebagai *Foreign Key* logis yang menghubungkan entitas CMS dengan logika bisnis kompleks pada sistem NestJS, memungkinkan sinkronisasi data antar-layanan mikro.

2. Struktur Konten Polimorfik
   Atribut `event_content` mengembalikan larik (*array*) objek dengan properti

diskriminator `__component`. Mekanisme ini memungkinkan *frontend* untuk merender komponen UI yang berbeda secara dinamis berdasarkan urutan data yang diterima.

3. Segregasi Metadata
   Pemisahan objek `seo` dari data konten utama mencerminkan prinsip *Separation of Concerns* (SoC), di mana data presentasi (konten) dipisahkan dari data meta (optimasi mesin pencari).

3.3.5 Pengembangan Layanan *Backend* (*Backend Services*)

Memasuki fase konstruksi logika bisnis pada minggu keenam, fokus kegiatan dipusatkan pada pengembangan layanan sisi peladen (*server-side services*) menggunakan kerangka kerja NestJS. Dalam arsitektur sistem yang telah dirancang, layanan *backend* ini berfungsi sebagai lapisan orkestrasi sentral yang bertanggung jawab mengelola interoperabilitas data antara antarmuka klien, sistem manajemen konten, dan infrastruktur persistensi data (*Supabase*).

Pengembangan modul ini dilakukan melalui pendekatan terstruktur yang dibagi menjadi tiga tahapan utama: perancangan alur logika, konfigurasi arsitektur, dan implementasi kode.

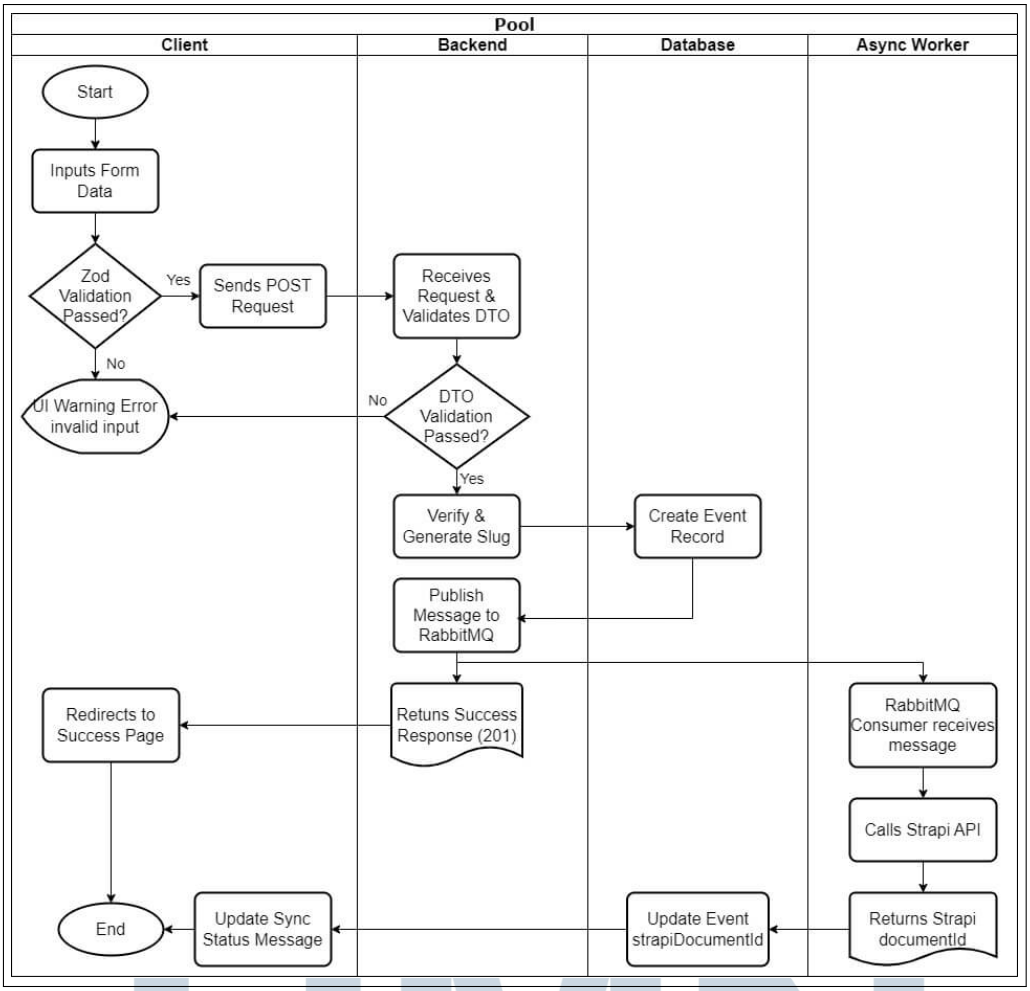
A Perancangan Alur Orkestrasi Proses *Backend*

Perancangan alur orkestrasi proses *backend* diawali dengan pemetaan alur kerja lintas komponen sistem yang terlibat dalam pengelolaan data acara. Tahap ini bertujuan untuk memastikan pemisahan tanggung jawab (*separation of concerns*) yang jelas antara lapisan antarmuka, logika bisnis inti, mekanisme persistensi data, serta proses sinkronisasi asinkron dengan sistem eksternal.

Untuk mencapai kejelasan tersebut, digunakan pendekatan *Cross-Functional Flowchart* atau Diagram Swimlane yang memungkinkan visualisasi interaksi antar entitas sistem secara kronologis dan terstruktur. Pendekatan ini dipilih karena alur manajemen acara melibatkan lebih dari satu jalur eksekusi, termasuk proses sinkronisasi latar belakang yang tidak bersifat blokir terhadap pengalaman pengguna.

Sebagaimana diilustrasikan pada Gambar 3.19, alur kerja melibatkan empat entitas utama, yaitu *Client*, *Backend*, *Database*, dan *Async Worker*, yang masing-masing memiliki peran spesifik dalam menjamin konsistensi data, responsivitas

sistem, serta skalabilitas layanan.



Gambar 3.19. Diagram Swimlane Alur Manajemen dan Sinkronisasi Data Acara

Berdasarkan diagram tersebut, mekanisme pemrosesan data dirancang dengan tahapan sebagai berikut:

1. Validasi Bertingkat (*Client* & *Backend*)
   Proses dimulai di sisi *Client* di mana pengguna memasukkan data formulir. Validasi awal dilakukan menggunakan pustaka *Zod* untuk memberikan umpan balik instan jika input tidak valid. Jika validasi lolos, permintaan dikirim ke *backend* yang kemudian melakukan validasi ulang menggunakan *DTO (Data Transfer Object)* untuk menjamin keamanan dan integritas tipe data sebelum diproses lebih lanjut.

2. Persistensi dan Percabangan Proses
   Setelah validasi *backend* terpenuhi, sistem memverifikasi dan menghasilkan

slug unik, lalu menyimpan data acara (*Create Event Record*) ke *Database*. Setelah data tersimpan, alur bercabang menjadi dua proses paralel:

- Jalur Respons Pengguna: *Backend* segera mengembalikan respons sukses (HTTP 201) ke *Client*, yang kemudian mengarahkan pengguna ke halaman sukses (*Success Page*) tanpa perlu menunggu sinkronisasi selesai.
- Jalur Sinkronisasi Latar Belakang
  Secara bersamaan, *backend* menerbitkan pesan ke *RabbitMQ*. Pesan ini ditangkap oleh *Async Worker*.

3. Sinkronisasi Data Eksternal: Pada jalur asinkron, *Async Worker* memanggil API Strapi untuk membuat entri konten. Setelah menerima balasan berupa `strapiDocumentId`, worker memperbarui rekaman di *Database* lokal. Proses diakhiri dengan pembaruan status sinkronisasi pada antarmuka *Client* (*Update Sync Status Message*).

B Inisialisasi dan Orkestrasi Layanan

Tahap pengembangan sistem diawali dengan proses inisialisasi aplikasi memanfaatkan modul `NestFactory`. Di dalam berkas `main.ts`, diterapkan mekanisme pengaturan layanan atau *Service Orchestration* guna menjamin stabilitas sistem. Langkah ini bertujuan untuk memastikan bahwa seluruh komponen infrastruktur pendukung telah siap sepenuhnya sebelum server dibuka untuk melayani permintaan HTTP.

Guna mengakomodasi kebutuhan sistem yang spesifik, strategi *bootstrap* tidak menggunakan metode inisialisasi standar, melainkan menerapkan logika orkestrasi bertahap sebagaimana didemonstrasikan pada Kode 3.3.

1. Konfigurasi Keamanan CORS (*Cross-Origin Resource Sharing*)

   Fungsi *whitelist* dinamis diterapkan guna membatasi akses API agar hanya dapat dijangkau oleh domain yang telah diizinkan melalui variabel lingkungan `CORS_ORIGIN`. Langkah preventif ini diambil untuk menutup celah keamanan dari serangan *Cross-Site Scripting* yang berasal dari sumber eksternal yang tidak terverifikasi.

2. Verifikasi Infrastruktur Krusial
   Sebelum server beroperasi penuh, sistem diprogram untuk melakukan koneksi eksplisit ke basis data (`PrismaService`), memverifikasi ketersediaan *Bucket* penyimpanan (`MediaService`), serta memastikan kesiapan topologi antrean pesan (`RabbitMQTopologyService`). Pendekatan ini bertujuan mencegah aplikasi berjalan dalam kondisi *"zombie"*, yaitu situasi di mana aplikasi tampak aktif namun sesungguhnya terputus dari layanan pendukung vitalnya.

3. Manajemen Siklus Hidup (*Graceful Shutdown*)
   Mekanisme *event listener* ditambahkan untuk merespons sinyal sistem seperti `SIGINT` dan `SIGTERM`. Fitur ini menjamin bahwa ketika aplikasi perlu dihentikan, misalnya saat proses pembaruan atau *deployment*, seluruh koneksi database dan proses yang sedang berjalan akan ditutup secara rapi demi menghindari risiko kerusakan atau korupsi data.

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // 1. Konfigurasi Keamanan CORS
  const allowedOrigins = process.env.CORS_ORIGIN?.split
  (',').map((o) =>
    o.trim().replace(/\/$/, '').toLowerCase(),
  );

  app.enableCors({
    origin: (origin, callback) => {
      if (!origin || allowedOrigins?.includes(origin.
  toLowerCase())) {
        callback(null, true);
      } else {
        callback(new Error('Not allowed by CORS'));
      }
    },
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE,OPTIONS',
    credentials: true,
```

```
    allowedHeaders: ['Content-Type', 'Authorization'],
    });

  // 2. Inisialisasi Layanan Pendukung
  const prisma = app.get(PrismaService);
  await prisma.$connect(); // Koneksi Database

  const mediaService = app.get(MediaService);
  await mediaService.ensureBucketExists(); // Cek Storage

  const rabbitTopology = app.get(RabbitMQTopologyService);
  await rabbitTopology.onModuleInit(); // Setup RabbitMQ

  // 3. Graceful Shutdown Handler
  process.on('SIGINT', async () => {
    await app.close();
    process.exit(0);
  });

  await app.listen(process.env.PORT || 3000);
}
bootstrap();
```

Kode 3.3: Logika Bootstrap Aplikasi pada main.ts

C Integrasi Infrastruktur Data dan Media

Setelah proses inisialisasi tuntas, tahap berikutnya adalah menghubungkan aplikasi dengan infrastruktur penyimpanan data. Pada fase ini, pola desain *Repository* diterapkan memanfaatkan *Prisma ORM* untuk mengelola interaksi basis data, serta layanan khusus yang dibangun untuk menangani penyimpanan objek atau berkas media.

1. Abstraksi Basis Data (Prisma Service)

Model data relasional didefinisikan secara terstruktur pada berkas `schema.prisma`. `PrismaService` kemudian diimplementasikan sebagai pengelola koneksi terpusat ke Supabase PostgreSQL guna menjamin efisiensi penggunaan sumber daya server.

2. Manajemen Penyimpanan Berkas (Media Service)

   Layanan ini dikembangkan untuk menangani fitur unggahan poster. Berbeda dengan metode biasa, sistem memvalidasi ketersediaan penyimpanan (*Bucket*) secara otomatis saat aplikasi dinyalakan (*bootstrap*) untuk mencegah kegagalan fungsi akibat kesalahan konfigurasi infrastruktur.

D Implementasi Struktur Data dan Validasi

Sesuai dengan alur kerja yang telah dirancang, proses implementasi diawali dengan penyusunan struktur data, baik di tingkat basis data maupun di sisi aplikasi.

D.1 Pemodelan Skema Relasional

Desain ERD yang telah disusun kemudian dituangkan ke dalam skema *Prisma ORM*. Seperti yang ditunjukkan pada Kode 3.4, entitas `Event` dilengkapi dengan kolom khusus, yakni `strapiDocumentId` dan `strapiSyncedAt`. Kedua atribut ini berperan sebagai penanda atau ”jangkar” sinkronisasi (*synchronization anchors*) yang bertugas memantau keselarasan data antara basis data utama dengan sistem CMS.

```
model Event {
  id              String       @id @default(cuid())
  slug            String       @unique
  title           String
  status          EventStatus  @default(DRAFT)
  startAt         DateTime

  // Field Integrasi Strapi (Headless CMS)
  strapiDocumentId     String?    @unique
  strapiSyncedAt       DateTime?
  strapiNeedsSync      Boolean    @default(true)

```

```
  // Relasi Media dan User
  poster      Media?     @relation("EventPoster", fields:
   [posterId], references: [id])
  createdBy User?      @relation("EventCreatedBy",
   fields: [createdById], references: [id])

  createdAt DateTime   @default(now())

  @@index([status, visibility]) // Optimasi query
}
```

Kode 3.4: Definisi Model Event dengan Field Sinkronisasi

D.2 Validasi Data Masukan (DTO)

Guna menjamin kualitas data sebelum masuk ke tahap pemrosesan sistem, diterapkan mekanisme validasi ketat menggunakan pustaka `class-validator`. Seperti terlihat pada Kode 3.5, setiap elemen input diperiksa format dan jenisnya secara teliti. Sebagai contoh, atribut `startAt` diwajibkan mengikuti format penanggalan standar (ISO), sedangkan `organizers` dipastikan berbentuk struktur data bertingkat atau *nested array*.

```
export class CreateEventDto {
  @IsString()
  title: string;

  @IsDateString() // Validasi format ISO 8601
  startAt: string;

  @IsOptional()
  @IsEnum(EventStatus)
  status?: EventStatus;

  // Validasi Relasi dan Struktur Nested
  @IsOptional()
  @IsArray()
  organizers: {
```

```
    organizerId: string;
    role: string;
    order: number;
  }[];

  @IsOptional()
  @IsArray()
  @IsString({ each: true })
  galleryMediaIds?: string[];
}
```

Kode 3.5: CreateEventDto dengan Validasi Bertingkat

E Implementasi Logika Bisnis Transaksional

Logika inti dari modul ini diimplementasikan pada layanan `EventsService`. Metode `create` (Kode 3.6) dirancang untuk menangani tiga tanggung jawab utama secara berurutan: generasi *slug* unik, persistensi data relasional, dan propagasi pesan asinkron.

```
async create(dto: CreateEventDto, userId?: string) {
  const { organizers, galleryMediaIds, title, slug, ...
   eventData } = dto;

  // 1. Generate Slug Unik untuk SEO Friendly URL
  const finalSlug = await generateUniqueSlug(this.
   prisma, title, slug);

  // 2. Transaksi Database dengan Relasi (Nested Writes
   )
  const event = await this.prisma.event.create({
    data: {
      ...eventData,
      title,
      slug: finalSlug,
      createdById: userId || null,
      startAt: new Date(dto.startAt),
```

```
      // Menyimpan relasi many-to-many organizer sekaligus
      organizers: organizers?.length ? {
        createMany: {
          data: organizers.map((o) => ({
            organizerId: o.organizerId,
            role: o.role || null,
            order: o.order || 0,
          })),
        },
      } : undefined,
    },
  });

  // 3. Publikasi Event ke Message Queue (Non-blocking)
  const msg: EventSyncMessage = {
    action: 'created',
    eventId: event.id,
    timestamp: new Date().toISOString(),
    event: { id: event.id, title: event.title, slug: event.slug },
  };

  await this.rabbitMqService.publish('event.created', msg);
  this.logger.log(`Published event.created for eventId=${event.id}`);

  return this.findOneById(event.id);
}
```

Kode 3.6: Logika Pembuatan Acara dan Publikasi RabbitMQ

Poin krusial pada implementasi di atas adalah penggunaan `this.rabbitMqService.publish`. Baris ini memastikan bahwa proses pembuatan konten di Strapi berjalan di latar belakang tanpa memblokir respons API ke pengguna, mewujudkan arsitektur *Event-Driven*.

F Eksposur *Secure Endpoint*

Langkah terakhir adalah menyediakan akses terhadap fungsi-fungsi sistem tersebut melalui antarmuka REST API yang terlindungi. Sebagaimana terlihat pada Kode 3.7, proteksi diterapkan menggunakan dekorator `@UseGuards(JwtAuthGuard)` untuk menjamin bahwa fitur pembuatan acara hanya dapat diakses oleh pengguna yang sudah terverifikasi. Selain itu, digunakan pula dekorator khusus `@CurrentUser` yang berfungsi membaca ID pengguna secara otomatis dari token digital (JWT). Mekanisme ini sangat penting untuk mencegah manipulasi data kepemilikan, karena identitas pengguna diambil langsung dari sistem autentikasi, bukan dari input manual.

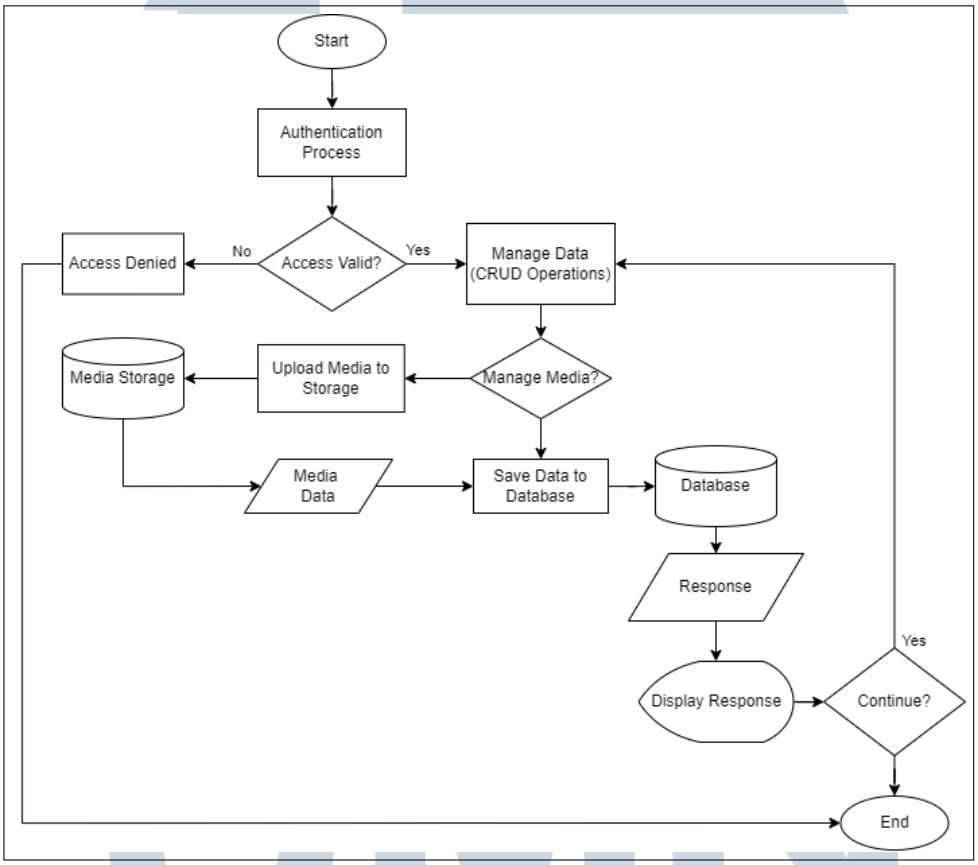
```
@UseGuards(JwtAuthGuard) // Proteksi Endpoint
@Post()
async create(@Body() body: CreateEventDto, @CurrentUser('id') userId) {
  try {
    return await this.eventService.create(body, userId);
  } catch (error) {
    throw new HttpException({
        success: false,
        message: error.message || 'Failed to create event',
      },
      HttpStatus.INTERNAL_SERVER_ERROR,
    );
  }
}
```

Kode 3.7: Kontroler API Pembuatan Acara

3.3.6 Perancangan Panel Administrasi (*Admin Panel*)

Fokus pengembangan selanjutnya beralih pada penyediaan perangkat internal atau *Internal Tooling* yang berperan sebagai pusat kendali operasional agensi. Sistem ini dibangun memanfaatkan teknologi *Next.js 16* dengan arsitektur

App Router dan pustaka *Shadcn UI* untuk menjamin konsistensi antarmuka. Guna memastikan integritas proses di dalam teknologi tersebut, disusunlah skema logika operasional terlebih dahulu sebelum melangkah ke tahap implementasi kode. Perancangan ini bertujuan memetakan interaksi administrator dengan data—mulai dari validasi sesi hingga umpan balik sistem—secara sistematis sebagaimana diilustrasikan pada Gambar 3.20.



Gambar 3.20. Diagram Alir Operasional Panel Admin

Berdasarkan diagram alur Operasional Admin, mekanisme sistem dirancang dengan tahapan prosedural sebagai berikut:

1. **Proses Autentikasi:** Sistem bertindak sebagai gerbang awal menggunakan mekanisme *NextAuth Proxy*. Sistem memvalidasi sesi pengguna; jika tidak valid, akses ditolak (*Access Denied*). Jika valid, administrator diizinkan masuk ke dasbor utama.

2. **Manajemen Data:** Administrator melakukan operasi CRUD (*Create, Read, Update, Delete*). Jika operasi melibatkan aset media (seperti gambar artis),

sistem akan memprioritaskan *Upload Media to Storage* terlebih dahulu sebelum menyimpan metadata ke basis data.

3. **Penyimpanan dan Umpan Balik:** Data disimpan ke basis data Supabase. Setelah berhasil, sistem memberikan *Response* visual secara langsung kepada pengguna untuk mengonfirmasi bahwa tindakan telah tersimpan, sebelum pengguna memutuskan untuk melanjutkan (*Continue*) atau mengakhiri sesi.

A Arsitektur Keamanan dan Komunikasi Data

Mengingat kompleksitas sistem terdistribusi, dirancanglah sebuah lapisan fondasi utama. Fokus dari lapisan ini adalah menangani proses autentikasi pengguna, menjaga keamanan akses rute, serta menstandarisasi alur komunikasi data ke sisi *backend* agar berjalan lebih teratur.

A.1 Autentikasi (NextAuth v4)

Implementasi autentikasi dilakukan menggunakan *NextAuth.js v4* dengan menerapkan strategi *Proxy Authentication*. Dalam skema ini, NextAuth difungsikan hanya sebagai perantara sesi yang bertugas meneruskan verifikasi data login ke API NestJS, tanpa harus membuka koneksi langsung ke basis data utama.

Logika autentikasi dikonfigurasi pada berkas `app/api/auth/[...nextauth]/route.ts`. Token JWT yang diterima dari *backend* dienkapsulasi ke dalam sesi enkripsi NextAuth melalui mekanisme *Callbacks*, sebagaimana ditunjukkan pada Kode 3.8.

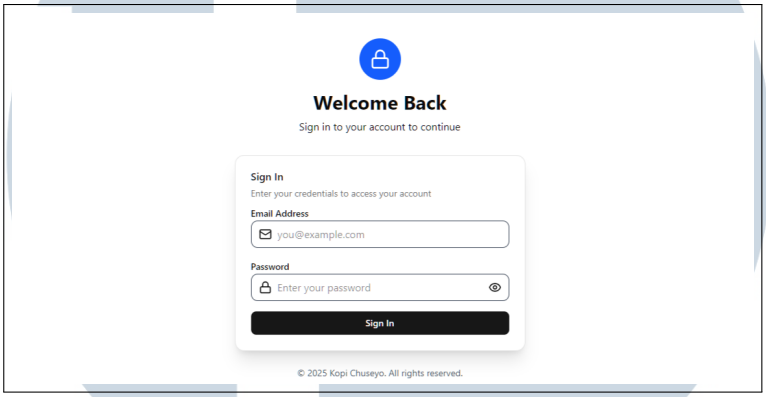
```
export const authOptions: AuthOptions = {
  providers: [
    Credentials({
      name: "credentials",
      credentials: { email: {}, password: {} },
      async authorize(credentials) {
        // Meneruskan kredensial ke NestJS Backend
        const res = await fetch(
          `${process.env.NEXT_PUBLIC_NEST_API_URL}/auth
  /login`,
          {
```

```
            method: "POST",
            headers: { "Content-Type": "application/
    json" },
            body: JSON.stringify(credentials),
          }
        );
        const data = await res.json();

        if (!res.ok) throw new Error(data.message || "
    Invalid Credentials");

        return {
          id: data.user.id,
          accessToken: data.accessToken, // Token JWT
    Backend
          role: data.user.role,
        };
      },
    }),
  ],
  callbacks: {
    // Mempersistenkan accessToken Backend ke dalam
    Token Sesi Next.js
    async jwt({ token, user }) {
      if (user) {
        token.accessToken = user.accessToken;
        token.role = user.role;
      }
      return token;
    },
    // Mengekspos accessToken ke sesi klien
    async session({ session, token }) {
      session.user.role = token.role;
      session.accessToken = token.accessToken;
      return session;
    },
```

```
    },
};
```

Kode 3.8: Konfigurasi Credentials Provider dan Callbacks pada NextAuth

Antarmuka halaman masuk (*Login Page*) yang terintegrasi dengan logika di atas ditampilkan pada Gambar 3.21.



Gambar 3.21. Antarmuka Halaman Autentikasi Admin Panel

A.2 Proteksi Rute Global (Middleware)

Lapisan keamanan preventif diterapkan menggunakan *Next.js Middleware* yang diatur dalam berkas `proxy.ts`. Fitur ini dioperasikan pada lingkungan *Edge Runtime* agar proses penyaringan akses tak sah dapat dilakukan dengan sangat cepat dan hambatan waktu (*latency*) seminimal mungkin. Sebagaimana terlihat pada Kode 3.9, logika sistem dirancang untuk menangani dua kondisi utama. Pertama, memblokir upaya akses tanpa izin ke halaman dasbor. Kedua, mencegah akses berulang yang tidak perlu ke halaman login bagi pengguna yang statusnya sudah terverifikasi.

```
import { NextResponse } from "next/server";
import type { NextRequest } from "next/server";
import { getToken } from "next-auth/jwt";

export default async function proxy(req: NextRequest) {
  const token = await getToken({
    req,
    secret: process.env.NEXTAUTH_SECRET,
```

```
  });

  const { pathname } = req.nextUrl;
  const publicPaths = ["/login", "/register"];
  const isPublic = publicPaths.some((path) => pathname.
   startsWith(path));

  // Skenario 1: Akses rute privat tanpa token ->
   Redirect ke Login
  if (!token && !isPublic) {
    const loginUrl = new URL("/login", req.url);
    return NextResponse.redirect(loginUrl);
  }

  // Skenario 2: Akses rute publik saat sudah login ->
   Redirect ke Dashboard
  if (token && isPublic) {
    const dashboardUrl = new URL("/", req.url);
    return NextResponse.redirect(dashboardUrl);
  }

  return NextResponse.next();
}
```

Kode 3.9: Logika Proteksi Rute dan Pengalihan pada Middleware

A.3 Integrasi API Terproteksi (Custom HTTP Client)

Guna memfasilitasi pertukaran data antara Panel Admin dan *backend* NestJS, diterapkanlah pola desain *Centralized HTTP Client Wrapper*. Pendekatan ini dipilih untuk mengatasi tantangan utama dalam arsitektur sistem terpisah, yakni keharusan untuk menyertakan token otorisasi secara konsisten pada setiap pengiriman data atau permintaan HTTP.

Sebagai langkah penyelesaian, dikembangkanlah kelas utilitas `ApiClient` (Kode 3.10) yang membungkus fungsi standar *Fetch API*. Modul ini dirancang menggunakan pola *Singleton* yang bertugas secara otomatis menyisipkan token sesi

NextAuth ke dalam *Header Authorization* setiap kali permintaan data dikirim.

```
class ApiClient {
  // Mengambil token dari sesi NextAuth secara dinamis
  private async getToken(): Promise<string | null> {
    if (!this.getTokenFn) {
      // Lazy loading modul Auth hanya saat dibutuhkan (Optimasi Bundle)
      const { getSession } = await import("next-auth/react");
      this.getTokenFn = async () => (await getSession())?.accessToken ?? null;
    }
    return await this.getTokenFn();
  }

  // Wrapper utama untuk semua permintaan HTTP
  private async request<T>(endpoint: string, options: RequestInit = {}): Promise<ApiResponse<T>> {
    const url = `${this.baseUrl}${endpoint}`;
    const headers = new Headers(options.headers);

    // 1. Suntikkan Token JWT (Bearer Token)
    const token = await this.getToken();
    if (token) {
      headers.set("Authorization", `Bearer ${token}`);
    }

    const config = { ...options, headers };
    const response = await fetch(url, config);

    // 2. Penanganan Otomatis Sesi Kedaluwarsa (401)
    if (!response.ok) {
      await this.handleUnauthorized(response);
    }

```

```
      return { success: true, data: await response.json()
      };
    }
}

export const apiClient = new ApiClient();
```

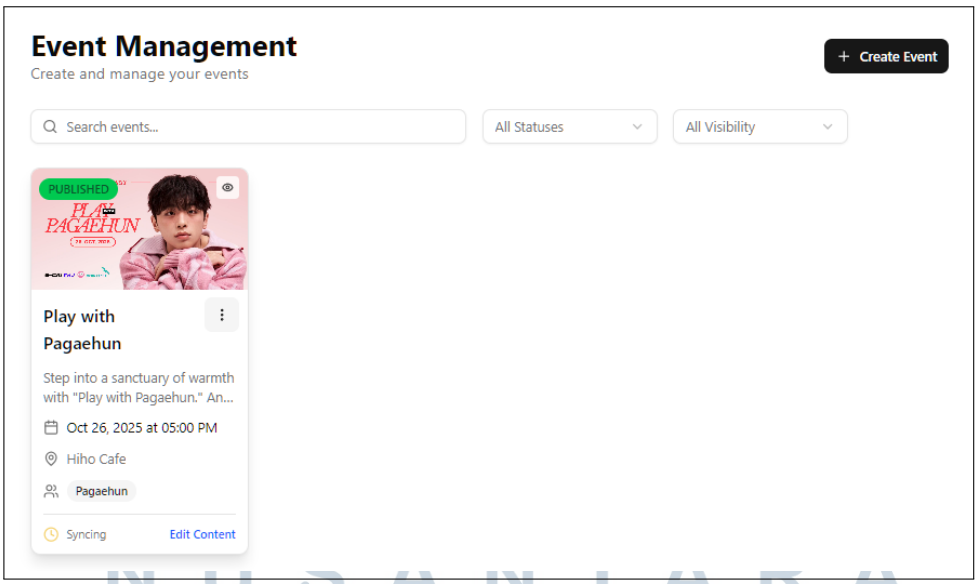
Kode 3.10: Implementasi Kelas ApiClient untuk Injeksi Token Otomatis

Mengacu pada Kode 3.10, mekanisme ini menawarkan keunggulan teknis berupa *Lazy Initialization*, di mana modul autentikasi hanya dimuat saat permintaan API pertama dilakukan, menjaga performa awal aplikasi tetap ringan.

B Implementasi Fitur: Modul Manajemen Acara

Setelah fondasi arsitektur terbentuk, fokus pengembangan dilanjutkan pada Modul *Event Management*. Modul ini dirancang untuk menangani kompleksitas data acara melalui antarmuka yang intuitif.

B.1 Visualisasi Data Real-time



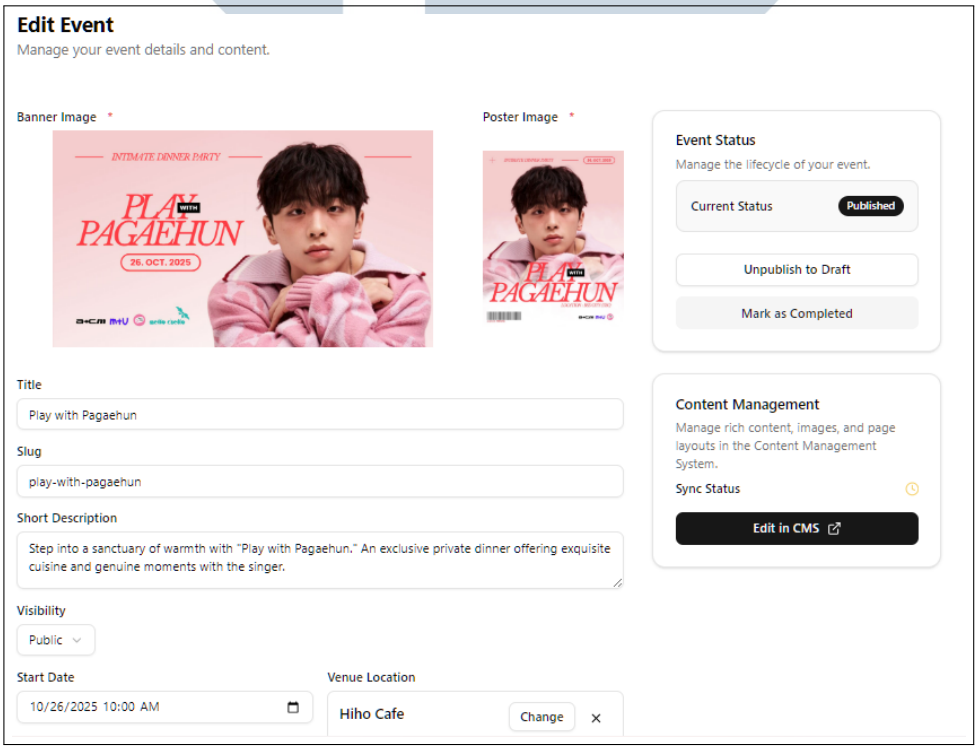
Gambar 3.22. Dasbor Daftar Acara dengan Indikator Status Sinkronisasi

Daftar acara divisualisasikan menggunakan komponen *Grid Card*. Sebagaimana terlihat pada Gambar 3.22, setiap kartu dilengkapi dengan indikator

status vital: label ”Published” untuk status publikasi dan ikon ”Syncing” di sudut kiri bawah untuk status sinkronisasi CMS. Indikator ini memungkinkan admin memantau kesehatan integrasi sistem secara *real-time*.

B.2 Validasi Formulir dan Konsistensi Skema

Integrasi antara pustaka *React Hook Form* dan skema validasi *Zod* dilakukan sebagai solusi untuk menangani input data yang kompleks. Penerapan aturan validasi di sisi antarmuka (*frontend*) ini kemudian diselaraskan dengan struktur DTO pada sisi server (*backend*). Melalui pendekatan tersebut, terciptalah keseragaman validasi atau *Schema Parity*, di mana standar pemeriksaan data yang sama berlaku secara konsisten di kedua lapisan sistem. Gambar 3.23 menampilkan implementasi formulir yang menggunakan komponen modular Shadcn UI. Sistem memberikan umpan balik validasi secara instan sebelum data dikirim ke server, meningkatkan pengalaman pengguna.



Gambar 3.23. Formulir Input Data Acara dengan Validasi Skema Zod

B.3 Abstraksi Logika Data (Custom Hooks)

Upaya memisahkan logika bisnis dari komponen tampilan dilakukan melalui penerapan pola *Custom React Hooks*. Sebagai realisasinya, dikembangkanlah modul `useEvents` (Kode 3.11) yang berperan sebagai jembatan penghubung antara elemen formulir dengan lapisan layanan API.

Hook ini tidak hanya menangani pengiriman data, tetapi juga mengelola status lokal aplikasi (*Local State Management*). Seperti terlihat pada fungsi `create`, setelah data berhasil dikirim ke *backend*, hook secara otomatis memperbarui *state* daftar acara tanpa perlu melakukan *refresh* halaman (*Optimistic UI Update*), memberikan pengalaman pengguna yang responsif.

```
export function useEvents(options: UseEventsOptions = {}) {
  const [events, setEvents] = useState<Event[]>([]);

  // Fungsi create yang dipanggil oleh Formulir
  const create = useCallback(async (data: CreateEventDto) => {
      // 1. Panggil API melalui Service Layer
      const newEvent = await eventsApi.create(data);

      // 2. Update State Lokal (tanpa reload halaman)
      if (!isSingleMode) {
        setEvents((prev) => [newEvent, ...(prev || [])]);
        setMeta((prev) => ({ ...prev, total: (prev?.total || 0) + 1 }));
      }

      return newEvent;
    }, [isSingleMode]);

  return { events, create, isLoading, ...others };
}
```

Kode 3.11: Logika Custom Hook useEvents untuk Manajemen Data

Dengan implementasi ini, alur pengiriman data menjadi terstruktur:

1. Pengguna menekan tombol simpan pada Formulir.
2. *React Hook Form* memvalidasi input menggunakan skema Zod.
3. Data valid dikirim ke fungsi `create` pada `useEvents`.
4. `useEvents` memanggil `apiClient` untuk meneruskan data ke NestJS.

C Pengembangan Modul Data Referensi dan Media

Melengkapi fitur utama manajemen acara, dikembangkan modul-modul pendukung yang berfokus pada pengelolaan entitas relasional, seperti Organizer dan Venue, serta sistem manajemen aset digital. Strategi pengembangan ini diterapkan dengan tujuan memisahkan ketergantungan data atau *data decoupling*, sekaligus meningkatkan efisiensi interaksi pengguna di dalam sistem.

C.1 Manajemen Sinkronisasi Data Formulir (*Dual State*)

Tantangan teknis muncul saat menghubungkan data relasional ke dalam formulir utama menggunakan pustaka *React Hook Form*. API backend membutuhkan senarai ID (*Array of Strings*) untuk relasi basis data, sementara Antarmuka Pengguna membutuhkan objek utuh untuk menampilkan pratinjau nama dan gambar tanpa perlu melakukan permintaan jaringan berulang.

Penerapan pola *Dual State Management* dipilih sebagai solusi untuk mengatasi kendala tersebut, dengan mekanisme kontrol yang mengandalkan validasi skema Zod. Seperti yang tercantum dalam Kode 3.12, aturan validasi ini memisahkan secara spesifik antara data yang berfungsi sebagai muatan kiriman atau *payload* (`organizerIds`) dan data yang ditujukan hanya untuk kebutuhan pratinjau visual (`organizers`).

```
const eventFormSchema = z.object({
  title: z.string().min(1, "Title is required"),

  // 1. Data Relasional untuk Payload API (Wajib)
  venueId: z.string().optional(),
  organizerIds: z.array(z.string()).optional(),

```

```
  // 2. Data Objek untuk Optimistic UI (Opsional)
  venue: z.any().optional(),
  organizers: z.array(z.any()).optional(),
});
```

Kode 3.12: Skema Validasi Zod untuk Relasi Data Ganda

Pola *Controlled Component* diterapkan melalui komponen pembungkus (*Wrapper*) untuk menjembatani logika sistem dengan antarmuka pengguna. Sebagaimana diperlihatkan pada Kode 3.13, mekanisme ini memungkinkan satu pemicu perubahan (`onChange`) untuk memperbarui dua sisi status atau *state* secara serentak.

```
<FormField
  control={form.control}
  name="organizers"
  render={({ field }) => (
    <FormItem>
      <FormLabel>Organizers</FormLabel>
      <FormControl>
        <OrganizerSelector
          value={field.value || []}
          onChange={(selectedOrganizers) => {
            // 1. Update State UI (Objek Utuh) untuk Preview
            field.onChange(selectedOrganizers);

            // 2. Ekstrak ID dan Update State Payload API
            const ids = selectedOrganizers.map((o) => o.organizerId);
            form.setValue("organizerIds", ids, { shouldValidate: true });
          }}
        />
      </FormControl>
      <FormMessage />
```

```
      </FormItem>
    )}
/>
```

Kode 3.13: Implementasi Controlled Component pada EventForm

C.2 Sistem Manajemen Media Ringan (*Native Implementation*)

Pengembangan fitur Pustaka Media (*Media Library*) dilakukan dengan pendekatan efisiensi, di mana ketergantungan pada pustaka pihak ketiga yang berukuran besar sengaja dihindari. Sebagai alternatif yang lebih ringan, mekanisme ini dibangun dengan memaksimalkan kemampuan bawaan *browser* atau *Native Browser API* (FormData dan Fetch).

Kendati hanya memanfaatkan API bawaan, aspek pengalaman pengguna tetap ditempatkan sebagai prioritas utama. Hal ini direalisasikan melalui penerapan mekanisme *Optimistic Feedback*, di mana antarmuka dirancang untuk memberikan umpan balik visual seketika saat proses pengunggahan berlangsung. Sebagaimana diperlihatkan pada Kode 3.14, fungsi pengunggahan bertugas menyusun objek `FormData` secara manual sekaligus mengontrol status `isUploading`. Langkah kontrol ini krusial untuk mencegah terjadinya pengiriman data ganda atau *double submission* yang tidak disengaja.

```
const handleUpload = async (e: React.ChangeEvent<
    HTMLInputElement>) => {
  const file = e.target.files?.[0];
  if (!file) return;

  try {
    setIsUploading(true); // 1. Set Loading State (UX
    Feedback)

    // 2. Konstruksi Payload Native
    const formData = new FormData();
    formData.append("file", file);

    // 3. Eksekusi Request ke Endpoint NestJS
    const res = await fetch("/api/media/upload", {
```

```
        method: "POST",
        headers: { Authorization: `Bearer ${session?.accessToken}` },
        body: formData,
      });

      if (!res.ok) throw new Error("Upload failed");
      await loadMedias();
      toast.success("Image uploaded successfully");

    } catch (error) {
      toast.error("Failed to upload image");
    } finally {
      setIsUploading(false);
    }
  };
```

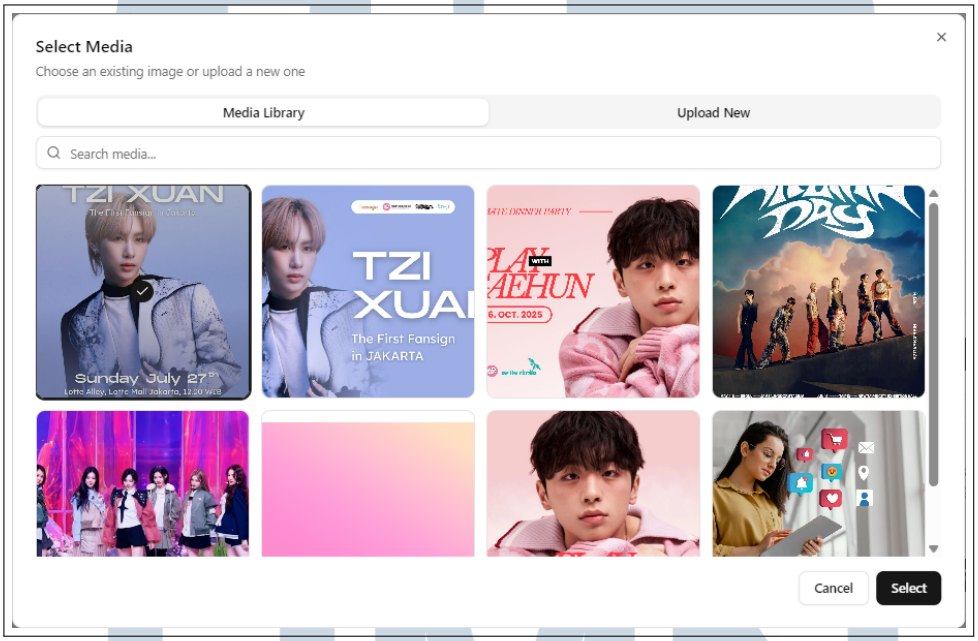
Kode 3.14: Logika Pengunggahan Native dengan Umpan Balik State

Selain itu, antarmuka galeri menerapkan strategi *Load All* dengan tampilan *Grid Layout*. Sebagaimana terlihat pada Kode 3.15 dan Gambar 3.24, sistem menggunakan *conditional rendering* untuk memberikan penanda visual (*ring*) pada gambar yang dipilih secara lokal.

```
<div className="grid grid-cols-3 md:grid-cols-4 gap-3">
  {filteredMedias.map((media) => (
    <div
      key={media.id}
      onClick={() => setSelectedMediaId(media.id)}
      className={cn(
        "relative aspect-square border-2 rounded-lg cursor-pointer",
        // Visual feedback saat dipilih: Border Primary + Icon Check
        selectedMediaId === media.id ? "border-primary ring-2" : "border-border"
      )}
```

```
      >
        <img src={media.url} className="object-cover w-
    full h-full" />
      </div>
    ))}
</div>
```

Kode 3.15: Implementasi Grid Media dengan Visual Feedback



Gambar 3.24. Antarmuka Pustaka Media dengan Tata Letak Grid

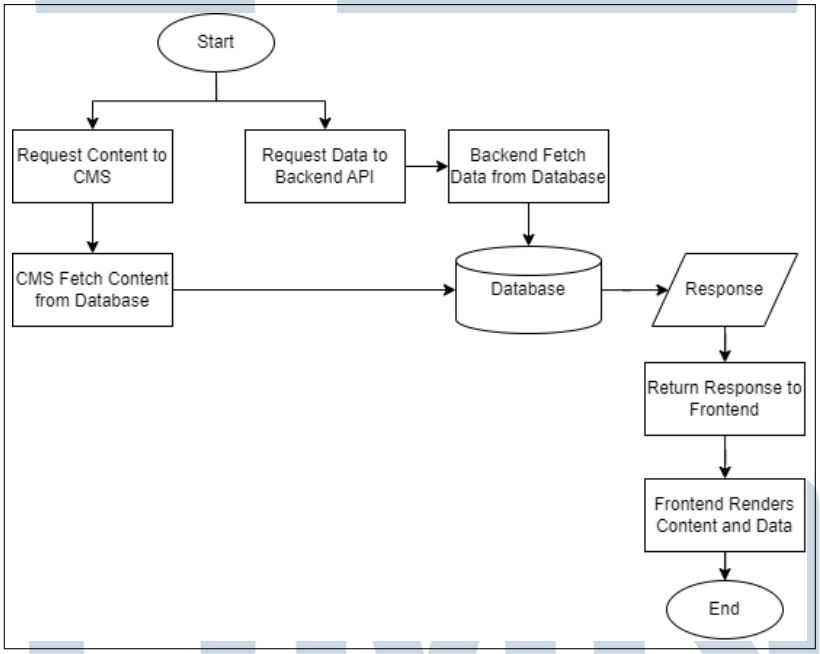
3.3.7 Pengembangan *Frontend* Website Perusahaan

Memasuki minggu kesembilan, fokus pengembangan bergeser pada konstruksi antarmuka publik (*Public-Facing Interface*) yang berfungsi sebagai etalase digital utama perusahaan. Fase ini bukan sekadar proses penerjemahan desain visual menjadi kode, melainkan rekayasa ulang arsitektur presentasi guna mendukung transformasi bisnis Kopi Chuseyo dari entitas F&B menjadi Agensi Digital profesional.

Paradigma *React Server Components* (RSC) dalam ekosistem Next.js 16 (*App Router*) diadopsi guna mencapai standar performa tinggi yang merefleksikan kompetensi teknis agensi. Keputusan arsitektur ini didasari oleh kebutuhan untuk menyeimbangkan pengalaman interaktif pengguna di sisi klien dengan kecepatan

kemunculan konten awal (*First Contentful Paint*) serta kinerja optimasi mesin pencari (*SEO*) yang superior. Kombinasi kemampuan ini merupakan peningkatan signifikan yang sebelumnya sulit dicapai secara optimal pada infrastruktur tema WordPress lama.

Untuk memperjelas bagaimana arsitektur *frontend* tersebut bekerja dalam menyajikan konten kepada pengguna, alur pengambilan dan penyajian data divisualisasikan melalui flowchart website publik pada Gambar 3.25. Flowchart ini menggambarkan proses permintaan data yang dilakukan oleh antarmuka website, baik terhadap sistem manajemen konten maupun layanan backend, hingga konten ditampilkan pada sisi pengguna.



Gambar 3.25. Flowchart Website Publik

Berdasarkan alur pada Gambar 3.25, website publik melakukan permintaan data ke dua sumber utama, yaitu sistem manajemen konten untuk kebutuhan informasi statis dan layanan backend untuk data aplikasi. Data yang diperoleh selanjutnya diproses pada sisi server dan dikirimkan kembali ke antarmuka untuk dirender menjadi konten yang siap dikonsumsi oleh pengguna akhir.

A Implementasi Arsitektur Antarmuka dan Komponen Modular

Penyusunan struktur arsitektur proyek pada berkas `layout.tsx` menjadi langkah awal dalam proses pengembangan. Berbeda dengan pendekatan

konvensional yang mengandalkan bantuan pustaka luar atau plugin tambahan untuk menangani SEO, konfigurasi optimasi mesin pencari pada sistem ini didefinisikan secara terprogram dengan memanfaatkan Metadata API bawaan dari Next.js.

Sebagaimana terlihat pada Kode 3.16, objek `metadata` dikonfigurasi secara eksplisit mencakup *Open Graph*, ikon, dan kata kunci strategis. Pendekatan ini memastikan setiap halaman memiliki standar indeksasi yang konsisten oleh mesin pencari.

Strategi *Font Optimization* turut diterapkan dengan memanfaatkan fitur `next/font`. Variabel font seperti `poppins`, `montserrat`, dan `bebasNeue` diintegrasikan langsung ke dalam tag `html` guna mencegah terjadinya pergeseran tata letak yang tidak stabil (*Cumulative Layout Shift*) saat halaman sedang dimuat.

```
export const metadata: Metadata = {
  title: {
    default: 'Kopi Chuseyo | K-Pop Digital Agency',
    template: '%s | Kopi Chuseyo',
  },
  description: 'Kopi Chuseyo helps brands and artists
   connect with Indonesian K-Pop fans...',
  keywords: ['kpop event organizer indonesia', 'kpop
   agency indonesia', ...],
  openGraph: {
    siteName: 'Kopi Chuseyo',
    locale: 'en_US',
    type: 'website',
  },
};

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html
      lang="en"
```

```
23        // Injeksi variabel font untuk optimasi CLS (Zero
      Layout Shift)
24        className={`${poppins.variable} ${montserrat.
    variable} ${bebasNeue.variable} bg-white`}
25      >
26        <SmoothScrolling />
27        <body className="flex min-h-screen justify-center
    ">
28          <div className="container">
29            <header>
30              <Navbar />
31            </header>
32
33            {/* Provider Transisi Halaman (Client-Side
    Transition) */}
34            <TransitionProvider>{children}</
    TransitionProvider>
35
36            <Footer />
37          </div>
38        </body>
39      </html>
40    );
41  }
```

Kode 3.16: Konfigurasi RootLayout dengan Metadata SEO dan Provider UI

Berdasarkan struktur JSX tersebut, pola *Persistent Layout* diterapkan dengan menempatkan komponen `Navbar` dan `Footer` di luar `TransitionProvider`. Pengaturan ini bertujuan agar elemen navigasi tetap pada posisinya (statis), sementara konten utama halaman (`children`) dapat menjalankan transisi animasi yang halus saat perpindahan rute berlangsung. Mekanisme tersebut sengaja dirancang untuk menciptakan pengalaman menjelajah yang responsif dan luwes, menyerupai sensasi menggunakan aplikasi seluler natif (*App-like Experience*).

Tahap pengembangan dilanjutkan dengan proses konversi desain menjadi kode (*Slicing*) setelah arsitektur dasar terbentuk. Metodologi *Component-Based Development* diterapkan dalam proses ini, di mana setiap bagian antarmuka

dipecah menjadi komponen modular yang dapat digunakan kembali (*Reusable Components*). Pendekatan tersebut memastikan konsistensi tampilan dan memudahkan pemeliharaan kode di masa mendatang.

Penataan gaya (*Styling*) dilakukan menggunakan Tailwind CSS dengan mengutamakan pendekatan *Mobile-First*. Melalui metode ini, responsivitas tampilan di berbagai ukuran perangkat, mulai dari telepon pintar hingga layar desktop lebar, dapat dipastikan berjalan optimal. Sebagai tambahan untuk menghadirkan kesan interaktif yang premium, pustaka Framer Motion turut diintegrasikan ke dalam sistem.

Pustaka Framer Motion diintegrasikan ke dalam sistem untuk memperkuat estetika visual agensi. Seperti yang ditunjukkan pada Kode 3.17, elemen *Hero Section* dilengkapi dengan efek animasi masuk (*entrance animation*) yang halus. Melalui penggunaan komponen `motion.div`, elemen-elemen tersebut dirancang untuk muncul secara bertahap (*staggered fade-in*) ketika halaman dibuka. Penerapan teknik ini bertujuan menciptakan kesan interaktif yang elegan tanpa memberikan beban berlebih pada kinerja *browser*.

```
// 1. Ekstraksi Logika Rotasi ke Custom Hook
const useAutoSlider = (length, duration = 8000) => {
  const [index, setIndex] = useState(0);
  const [progress, setProgress] = useState(0);

  useEffect(() => {
    const timer = setInterval(() => {
      // Logika kalkulasi progress bar dan rotasi index
      setProgress((prev) => Math.min((prev + 50 /
   duration) * 100, 100));
      // ... logika next slide
    }, 50);
    return () => clearInterval(timer);
  }, [length]);

  return { index, progress, setIndex };
};

// 2. Definisi Varian Animasi (Declarative Animation)
```

```
const ANIMATION = {
  container: {
    hidden: { opacity: 0 },
    visible: { opacity: 1, transition: {
   staggerChildren: 0.15 } }
  },
  item: {
    hidden: { opacity: 0, y: 20 },
    visible: { opacity: 1, y: 0, transition: { duration
   : 0.6 } }
  }
};

// 3. Komponen Utama (Presentational)
export default function HeroSection({ data }) {
  const { hero_headline, hero_images } = data;

  return (
    <SectionContainer className="relative flex min-h
   -[90vh] ...">
      {/* Bagian Teks dengan Animasi Bertingkat */}
      <motion.div
        variants={ANIMATION.container}
        initial="hidden"
        whileInView="visible"
        viewport={{ once: true }}
        className="relative z-10 flex flex-col justify-
   center"
      >
        <motion.h1 variants={ANIMATION.item} className=
   "text-5xl font-bold ...">
          {hero_headline}
        </motion.h1>

        <motion.div variants={ANIMATION.item}>
          <Button href={data.hero_cta_link}>Explore
```

```
    Services </Button>
50              </motion.div>
51            </motion.div>
52
53            {/* Bagian Gambar dengan Logika Hook */}
54            <HeroImageSection images={hero_images} />
55          </SectionContainer>
56        );
57    }
```

Kode 3.17: Implementasi Hero Section dengan Custom Hook dan Animasi

B Integrasi Konten Dinamis dan Renderasi Polimorfik

Tantangan utama dalam pengembangan antarmuka publik adalah menangani struktur data yang sangat dinamis dari Strapi CMS, khususnya pada fitur *Dynamic Zone*. Berbeda dengan kolom basis data tradisional yang kaku, fitur ini memungkinkan admin menyusun tata letak halaman secara bebas menggunakan berbagai kombinasi komponen.

Pola arsitektur *Polymorphic Rendering* diimplementasikan sebagai solusi untuk menangani kondisi tersebut, dengan dukungan strategi pengambilan data bersarang (*Deep Population*). Melalui pendekatan ini, struktur data yang kompleks dari berbagai sumber dapat ditarik secara mendalam hingga ke level relasi terkecil. Hal tersebut memungkinkan antarmuka untuk menampilkan konten yang dinamis dan beragam secara fleksibel, tanpa harus melakukan banyak permintaan data secara terpisah ke server.

B.1 Strategi Pengambilan Data Bersarang (*Deep Population*)

Data relasi seperti galeri gambar tidak dikembalikan secara otomatis oleh Strapi demi efisiensi performa. Sebagai solusinya, disusunlah parameter *query* khusus menggunakan notasi *LHS Bracket*.

Sebagaimana diperlihatkan pada Kode 3.18, penggunaan parameter `populate[event_content][populate]=*` menginstruksikan API untuk menarik seluruh data *Dynamic Zone* beserta medianya dalam satu permintaan tunggal. Selain itu, diterapkan pula mekanisme *Incremental Static Regeneration* (ISR)

dengan revalidasi tiap 60 detik. Strategi ini bertujuan menjaga ketersediaan data terbaru tanpa membebani kinerja server secara berlebihan.

```
export async function fetchStrapiEvent(slug: string) {
  try {
    // Mengambil data Event berdasarkan Slug
    // Menggunakan teknik Nested Populate untuk Dynamic Zone
    const res = await fetch(
      `${process.env.NEXT_PUBLIC_STRAPI_URL}/api/events?filters[event_slug][$eq]=${slug}&populate[event_content][populate]=*&populate[seo][populate]=*`,
      {
        headers: {
          Authorization: `Bearer ${process.env.STRAPI_READ_ONLY_API_KEY}`,
        },
        // Strategi ISR: Revalidasi cache setiap 60 detik
        next: { revalidate: 60 },
      }
    );

    const data = await res.json();
    return data.data[0] || null;
  } catch (error) {
    console.error("Error fetching event:", error);
    return null;
  }
}
```

Kode 3.18: Logika Fetching dengan Deep Population Query

B.2 Mekanisme Renderasi Blok Dinamis

Informasi yang diterima dari API berupa kumpulan data (*array*) objek yang dilengkapi properti diskriminator unik bernama `__component`. Guna mengolah data tersebut, dikembangkanlah komponen `BlockRenderer` yang bertugas sebagai pusat kendali distribusi (*Dispatcher*).

Komponen ini difungsikan untuk memetakan string diskriminator dari Strapi ke dalam komponen React yang relevan secara *runtime*. Sebagaimana diperlihatkan pada Kode 3.19, logika `switch-case` diterapkan untuk menangani berbagai tipe blok konten, mulai dari Paragraf Dua Kolom dan Galeri Gambar hingga Teks Kaya (*Rich Text*). Melalui mekanisme ini, setiap bagian konten dapat ditampilkan secara dinamis sesuai dengan jenis data yang dikirimkan oleh server.

```
export default function BlockRenderer({ content, className }: Props) {
  // Validasi keberadaan konten dynamic zone
  if (!content || !Array.isArray(content)) return null;

  return (
    <div className={className}>
      {content.map((block: any, index: number) => {
        // Mapping komponen berdasarkan properti '__component'
        switch (block.__component) {
          case 'two-columns.paragraph-image':
            return <TwoColumnsParagraphImage key={index} data={block} />;

          case 'two-columns.image-paragraph':
            return <TwoColumnsImageParagraph key={index} data={block} />;

          case 'image-gallery.image-gallery':
            return <ImageGallery key={index} data={block} />;

```

```
          case 'blog-components.rich-text':
            return (
              <div key={index} className="prose prose-lg prose-neutral my-8">
                <StrapiBlockRenderer content={block.content} />
              </div>
            );

          default:
            console.warn(`Unknown component: ${block.__component}`);
            return null;
        }
      })}
    </div>
  );
}
```

Kode 3.19: Implementasi Pola Dispatcher pada BlockRenderer

Melalui pendekatan ini, aplikasi memiliki fleksibilitas tinggi. Jika di masa depan diperlukan jenis tampilan baru, pengembang cukup menambahkan *case* baru pada `BlockRenderer` tanpa perlu mengubah struktur basis data inti.

C Optimasi Performa dan SEO Dinamis

Serangkaian optimasi dilakukan sebagai tahap akhir pengembangan antarmuka publik guna memastikan visibilitas website di mesin pencari serta kecepatan pengalaman pengguna. Fokus utama pada fase ini diarahkan pada penerapan Metadata Dinamis dan optimasi aset visual untuk memenuhi standar kualitas pengalaman web atau *Core Web Vitals*.

C.1 Implementasi SEO Dinamis dan Data Terstruktur

Berbeda dengan halaman statis, kebutuhan metadata pada halaman detail acara (*Event Details*) bersifat unik untuk setiap kontennya. Fungsi

`generateMetadata` dari Next.js dimanfaatkan untuk menyisipkan tag *meta* secara dinamis, seperti judul, deskripsi, dan *OpenGraph*, berdasarkan data yang diperoleh dari Strapi.

Selain itu, sebagaimana diperlihatkan pada Kode 3.20, *Structured Data* (JSON-LD) turut disertakan ke dalam dokumen. Skrip ini berfungsi membantu mesin pencari seperti Google dalam memahami konteks konten secara mendalam, seperti tanggal acara, lokasi, hingga harga tiket. Informasi tersebut nantinya dapat ditampilkan sebagai *Rich Snippets* guna meningkatkan daya tarik visual pada hasil pencarian.

```
// 1. Generate Metadata Dinamis (Server-Side)
export async function generateMetadata({ params }): Promise<Metadata> {
  const { slug } = await params;
  const strapiEvent = await fetchStrapiEvent(slug);
  const { seo } = strapiEvent;

  return {
    title: seo.metaTitle,
    description: seo.metaDescription,
    openGraph: {
      title: seo.metaTitle,
      description: seo.metaDescription,
    },
    // Konfigurasi Canonical URL untuk mencegah duplikasi konten
    alternates: { canonical: seo.canonicalURL },
  };
}

// 2. Injeksi JSON-LD pada Komponen Page
export default async function EventPage({ params }) {
  const strapiEvent = await fetchStrapiEvent(params.slug);

  return (
```

```
    <>
      {/* Script untuk Rich Snippets (Schema.org) */}
      {strapiEvent?.seo?.structuredData && (
        <script
          type="application/ld+json"
          dangerouslySetInnerHTML={{
            __html: JSON.stringify(strapiEvent.seo.
    structuredData),
          }}
        />
      )}
      {/* ... render konten halaman */}
    </>
  );
}
```

Kode 3.20: Implementasi Metadata Dinamis dan Injeksi JSON-LD

Komponen `Image` dari Next.js diimplementasikan untuk menjaga stabilitas skor *Core Web Vitals*, khususnya pada metrik kecepatan pemuatan elemen utama atau *Largest Contentful Paint* (LCP). Detail teknis konfigurasi aset visual prioritas tinggi ini diperlihatkan pada Kode 3.21.

Dalam implementasi tersebut, properti `priority` disematkan secara eksplisit pada komponen *Banner* untuk memaksa sistem memprioritaskan pengunduhan gambar dan menonaktifkan fitur *lazy-loading*. Bersamaan dengan itu, atribut dimensi lebar dan tinggi didefinisikan secara statis guna memreservasi ruang layar sebelum gambar dimuat sepenuhnya, sebuah langkah krusial untuk mencegah pergeseran tata letak yang tidak stabil (*Cumulative Layout Shift*).

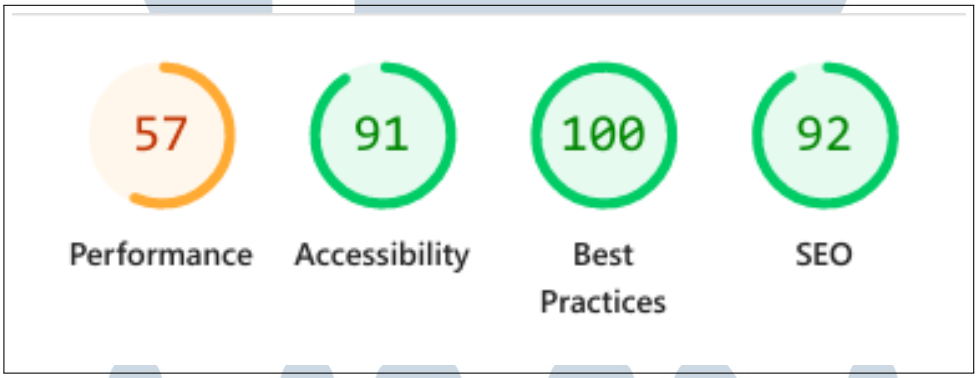
```
<div className="relative w-full">
  <Image
    src={event.bannerUrl || '/placeholder.png'}
    alt={`${event.title} Banner`}
    width={1920}
    height={1080}
    // Mencegah Layout Shift (CLS) dan memacu LCP
    className="h-auto w-full object-contain"
```

```
      priority={true}
    />
  </div>
```

Kode 3.21: Optimasi Banner Utama dengan Prioritas LCP

C.2 Evaluasi Metrik Web Vital dan SEO

Setelah tahap implementasi selesai, pengujian audit dilakukan menggunakan perangkat lunak *Google Lighthouse* (versi 13.0.1) pada lingkungan produksi (*Vercel Deployment*). Pengujian ini difungsikan untuk mengukur efektivitas optimasi yang telah diterapkan terhadap standar kualitas pengalaman pengguna atau *Core Web Vitals*.



Gambar 3.26. Hasil Audit Lighthouse: Skor SEO Tinggi dengan Catatan pada Performa

Berdasarkan hasil pengujian yang ditampilkan pada Gambar 3.26, website berhasil mencapai skor SEO sebesar 92. Pencapaian ini mengindikasikan bahwa implementasi *Metadata API* dinamis, struktur semantik HTML, dan penerapan atribut *accessibility* dasar telah berjalan dengan sangat baik, sehingga konten website mudah dirayapi (*crawlable*) oleh mesin pencari.

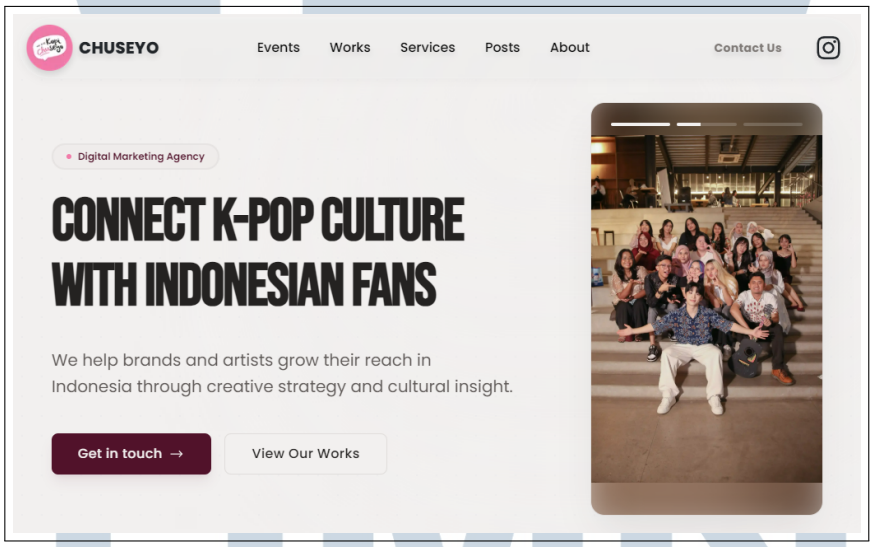
Kendati demikian, tercatat adanya titik temu (*trade-off*) antara kualitas visual dan metrik performa. Penggunaan aset visual beresolusi tinggi serta animasi interaktif dari Framer Motion guna memenuhi standar estetika ”Agensi Digital Premium” memberikan beban tambahan pada proses utama (*Main Thread*), yang berdampak pada penurunan skor performa. Hal ini menjadi catatan evaluasi bahwa diperlukan strategi optimasi lanjutan di masa mendatang, seperti penerapan pemecahan kode (*Code Splitting*) yang lebih agresif serta pemanfaatan jaringan pengiriman konten (*Content Delivery Network*) eksternal. Langkah tersebut krusial

untuk menjaga keseimbangan antara kejernihan visual (*fidelity*) dan kecepatan muat halaman.

D Realisasi Antarmuka Pengguna

Sebagai tahap akhir, hasil implementasi antarmuka disajikan untuk memvalidasi kesesuaian antara kode program dengan rancangan desain. Berikut adalah tiga tampilan utama yang merepresentasikan keberhasilan sistem dalam menangani identitas visual, struktur data dinamis, dan adaptabilitas perangkat.

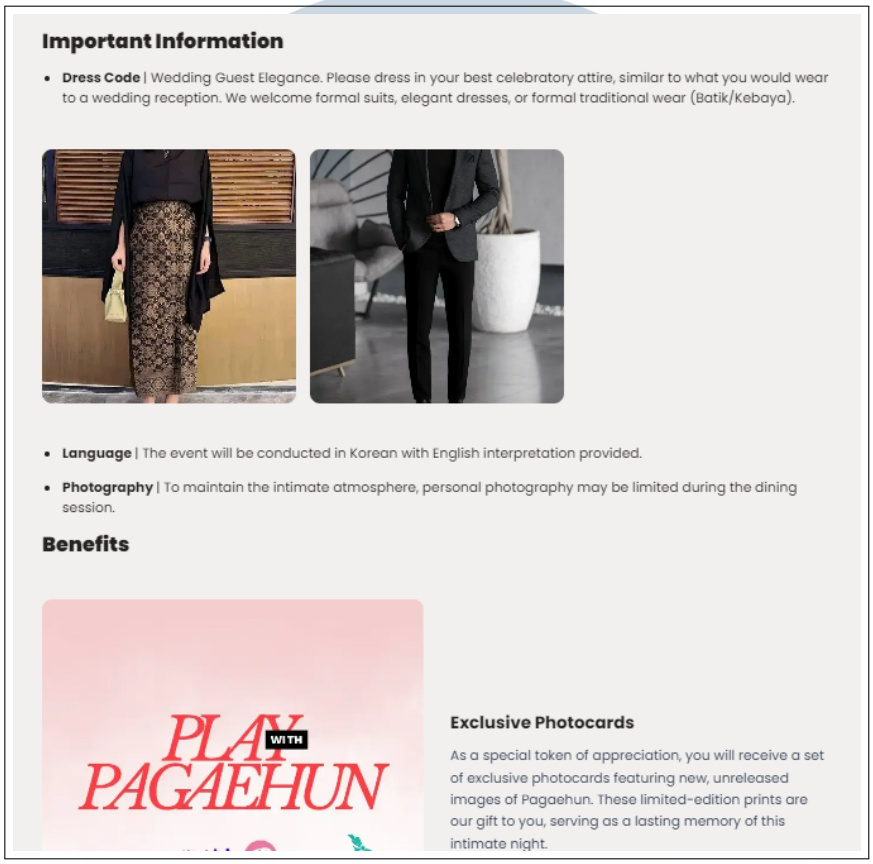
D.1 Implementasi Identitas Visual Baru



Gambar 3.27. Tampilan Hero Section: Implementasi Identitas Visual Agensi Digital

Pada halaman utama, transformasi identitas agensi direalisasikan melalui penerapan tipografi tegas dan skema warna monokromatis. Sebagaimana terlihat pada Gambar 3.27, komponen *Hero Section* berhasil merender data dinamis dari API dengan tetap mempertahankan tata letak yang presisi dan estetika profesional.

D.2 Renderasi Konten Polimorfik



Gambar 3.28. Tampilan Detail Portfolio: Hasil Renderasi Blok Konten Dinamis

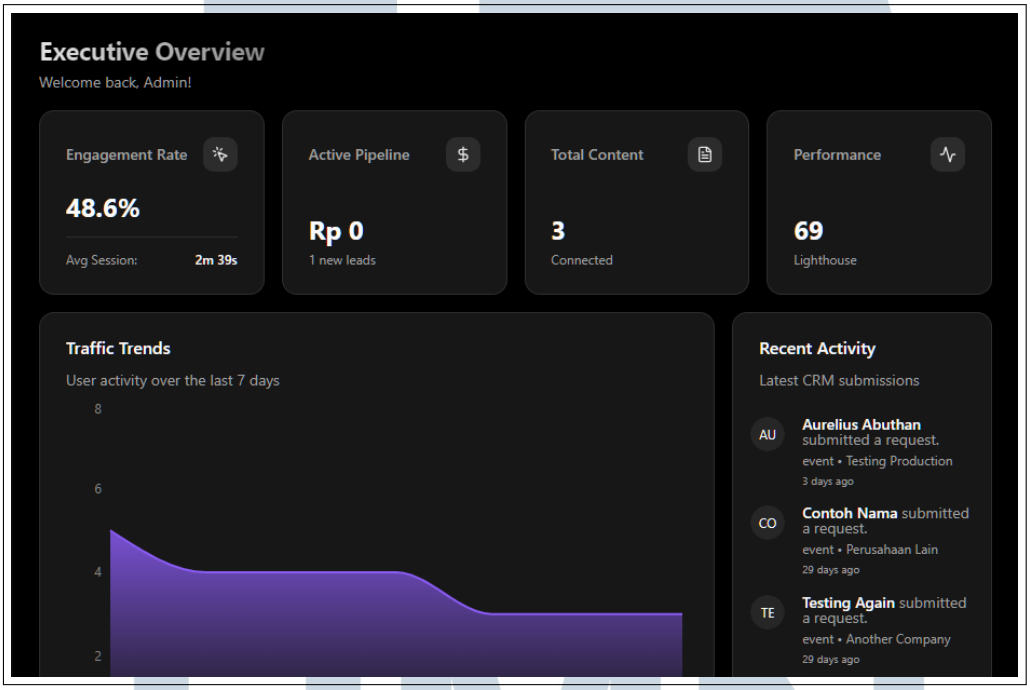
Validasi teknis terhadap komponen `BlockRenderer` ditunjukkan pada halaman detail acara. Gambar 3.28 memperlihatkan kemampuan sistem dalam menyusun berbagai tipe blok konten—mulai dari teks kaya (*Rich Text*) hingga galeri media—menjadi satu tampilan artikel yang kohesif tanpa merusak struktur tata letak.

3.3.8 Pengembangan *Dashboard* Analitik Terintegrasi

Setelah modul operasional pada Panel Administrasi selesai dikembangkan, fokus pengembangan diarahkan pada penyediaan sarana analisis kinerja aplikasi secara menyeluruh. Pada tahap ini, dibangun sebuah *Dashboard* Analitik yang berfungsi sebagai *Executive Overview*, yang menyajikan informasi perilaku pengguna, performa teknis aplikasi, serta indikator bisnis utama dalam satu antarmuka terpadu.

Dashboard ini dirancang dengan pendekatan *data-driven*, di mana seluruh informasi diturunkan langsung dari sumber data aktual. Integrasi dilakukan terhadap layanan pihak ketiga, yaitu Google Analytics 4 dan Google Lighthouse, serta dikombinasikan dengan data internal sistem, sehingga menghasilkan gambaran kondisi aplikasi yang komprehensif dan dapat digunakan sebagai dasar pengambilan keputusan strategis.

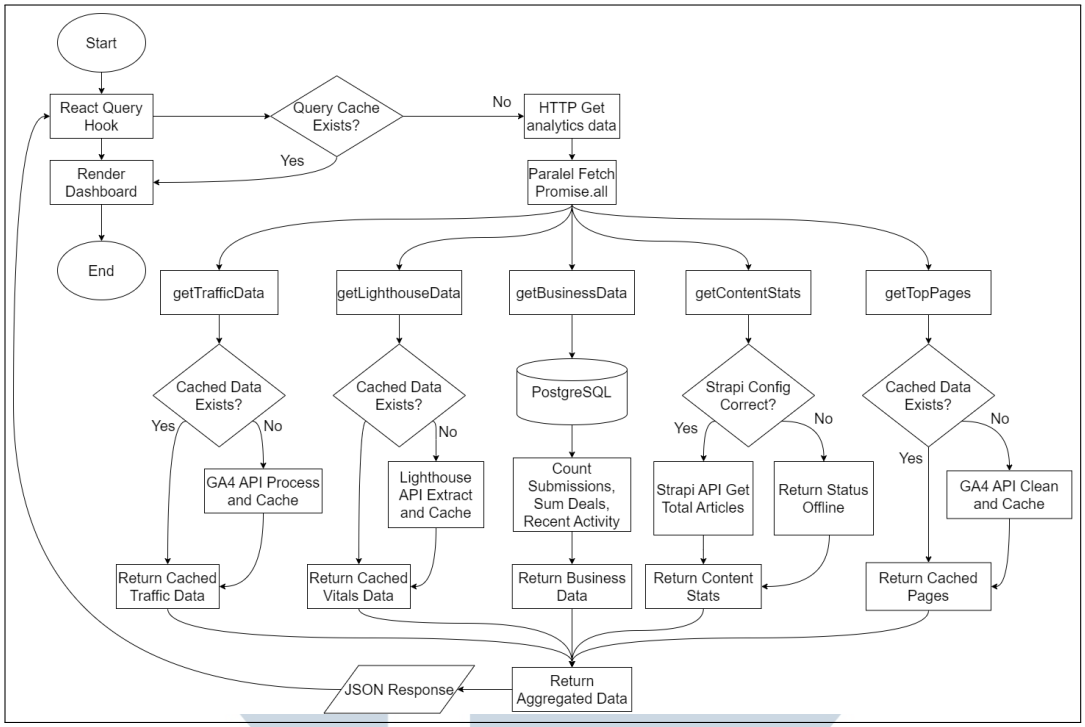
Antarmuka utama *Dashboard* Analitik ditampilkan pada Gambar 3.29.



Gambar 3.29. Tampilan *Dashboard* Analitik Terintegrasi

A Arsitektur Pengambilan dan Agregasi Data Analitik

Mengingat sumber data analitik yang bersifat heterogen dan terdistribusi, diperlukan sebuah arsitektur pengambilan data yang terstandarisasi dan efisien. Oleh karena itu, seluruh proses pengambilan, pengolahan, dan agregasi data analitik ditempatkan sepenuhnya pada sisi *backend*, sementara sisi *frontend* hanya berperan sebagai penyaji data.



Gambar 3.30. Flowchart Arsitektur Pengambilan dan Agregasi Data Analitik

Gambar 3.30 menggambarkan alur pengambilan dan pengolahan data pada *Dashboard* Analitik. Proses dimulai dari permintaan data yang dipicu oleh antarmuka *Dashboard* pada sisi *frontend*, kemudian diteruskan ke layanan *backend* melalui satu *endpoint* agregasi.

Pada sisi *backend*, layanan analitik bertanggung jawab untuk mengambil data dari berbagai sumber, yaitu Google Analytics 4, Google Lighthouse, serta basis data internal sistem. Seluruh data tersebut diproses dan dikonsolidasikan dalam format terstandarisasi sebelum dikirimkan kembali ke sisi *frontend* untuk ditampilkan dalam bentuk visualisasi analitik.

Pendekatan ini diterapkan untuk menjaga keamanan kredensial API, mengurangi kompleksitas logika di sisi klien, serta memastikan konsistensi format data yang diterima oleh komponen antarmuka. Seluruh data analitik dikonsolidasikan melalui satu *endpoint* agregasi, sebagaimana diimplementasikan pada Kode 3.22.

```
@Get('dashboard')
async getDashboard() {
  const { traffic, vitals, business, content, topPages
  } =
```

```
    await this.analyticsService.getAggregatedStats();

  return {
    success: true,
    data: { traffic, vitals, business, content,
    topPages },
    message: 'Analytics dashboard fetched successfully'
    ,
  };
}
```

Kode 3.22: Endpoint Agregasi Data Dashboard Analitik

B Integrasi Google Analytics 4 untuk Analisis Perilaku Pengguna

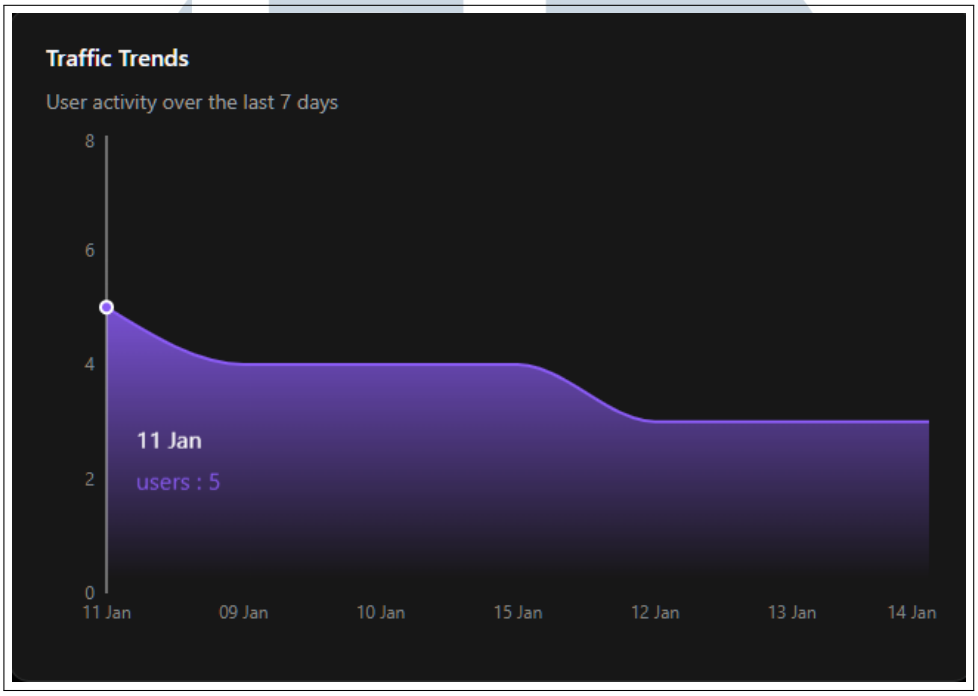
Google Analytics 4 (GA4) diintegrasikan untuk memperoleh data perilaku pengguna yang berinteraksi dengan aplikasi. Integrasi ini dilakukan menggunakan Google Analytics Data API yang dieksekusi secara *server-side* melalui layanan *backend* berbasis NestJS.

Metrik utama yang dikumpulkan meliputi jumlah pengguna aktif, jumlah tampilan halaman, tingkat keterlibatan (*engagement rate*), serta durasi sesi rata-rata. Proses pengambilan dan normalisasi data GA4 diimplementasikan sebagaimana ditunjukkan pada Kode 3.23.

```
const [report] = await analyticsClient.runReport({
  property: `properties/${propertyId}`,
  dateRanges: [{ startDate: '7daysAgo', endDate: 'today
  ' }],
  metrics: [
    { name: 'activeUsers' },
    { name: 'screenPageViews' },
    { name: 'engagementRate' },
    { name: 'averageSessionDuration' }
  ],
  dimensions: [{ name: 'date' }]
});
```

Kode 3.23: Pengambilan dan Pemrosesan Data Trafik dari Google Analytics 4

Visualisasi data trafik pengguna dalam bentuk grafik tren ditampilkan pada Gambar 3.31.



Gambar 3.31. Visualisasi Tren Aktivitas Pengguna Berdasarkan Data GA4

C Integrasi Google Lighthouse untuk Evaluasi Performa Aplikasi

Selain analisis perilaku pengguna, *Dashboard* Analitik juga mengintegrasikan Google Lighthouse melalui PageSpeed Insights API untuk mengevaluasi performa teknis aplikasi web. Fokus evaluasi diarahkan pada metrik *Core Web Vitals* yang secara langsung memengaruhi pengalaman pengguna.

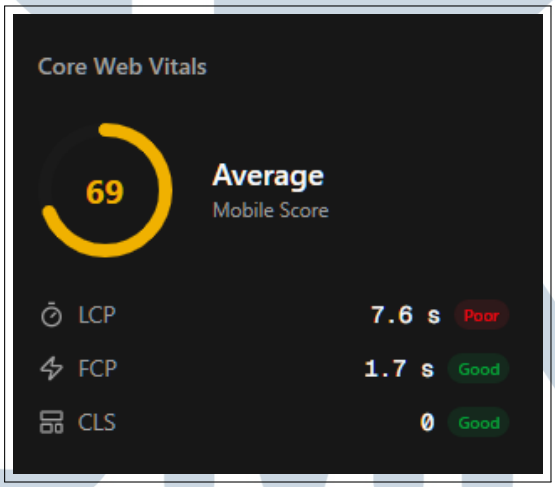
Proses pemanggilan API Lighthouse dan ekstraksi metrik performa diimplementasikan sebagaimana ditunjukkan pada Kode 3.24.

```
1 const response = await httpService.get(endpoint, {
    params });
2 const lighthouseResult = response.data?.
    lighthouseResult;
3
```

```
return {
  performanceScore: Math.round(
    lighthouseResult.categories.performance.score * 100
  ),
  lcp: lighthouseResult.audits['largest-contentful-
   paint']?.displayValue,
  cls: lighthouseResult.audits['cumulative-layout-shift
   ']?.displayValue,
  fcp: lighthouseResult.audits['first-contentful-paint'
   ]?.displayValue,
};
```

Kode 3.24: Pengambilan Data Core Web Vitals dari Google Lighthouse

Ringkasan metrik performa aplikasi ditampilkan pada antarmuka *Dashboard* sebagaimana diperlihatkan pada Gambar 3.32.



Gambar 3.32. Ringkasan Core Web Vitals dan Skor Performa Aplikasi

D Manajemen State dan Penyajian Data pada Antarmuka

Pada sisi *frontend*, pengelolaan data *Dashboard* Analitik dilakukan menggunakan pola *Custom React Hook* yang dipadukan dengan pustaka *React Query*. Implementasi ini bertujuan untuk memisahkan logika pengambilan data dari komponen tampilan, serta memanfaatkan fitur *caching* dan *error handling* bawaan.

Logika pengambilan data *Dashboard* diimplementasikan melalui hook `useDashboardStats`, sebagaimana ditunjukkan pada Kode 3.25.

```
export function useDashboardStats() {
  return useQuery({
    queryKey: ['dashboard-stats'],
    queryFn: async () => {
      const res = await apiClient.get('/analytics/
    dashboard');
      return res.data;
    },
    staleTime: 1000 * 60 * 5,
    retry: 2,
  });
}
```

Kode 3.25: Custom Hook useDashboardStats untuk Pengambilan Data Analitik

3.3.9 Deployment dan Konfigurasi Infrastruktur

Siklus pengembangan perangkat lunak diakhiri dengan tahap rilis produksi (*Deployment*). Mengingat aplikasi ini terdiri dari beberapa layanan yang terpisah (*Decoupled Architecture*), topologi infrastruktur hibrida dirancang dengan menggabungkan fleksibilitas *Infrastructure as a Service* (IaaS) dan kemudahan *Platform as a Service* (PaaS). Kombinasi ini dipilih untuk menciptakan ekosistem penyebaran aplikasi yang tangguh namun tetap mudah dikelola.

Arsitektur deployment dibagi menjadi tiga segmen utama: layanan komputasi *backend* pada Google Cloud Platform, manajemen data terpusat menggunakan Supabase, dan distribusi antarmuka *frontend* melalui jaringan Vercel.

A Manajemen Infrastruktur Backend (Google Compute Engine)

Lingkungan Google Compute Engine (VM) dipilih sebagai basis layanan *backend* (NestJS dan Strapi). Pemilihan infrastruktur berbasis mesin virtual (*Virtual Machine*) ini memberikan kontrol penuh terhadap konfigurasi sistem operasi serta jaringan, yang sangat krusial untuk melakukan penyesuaian sistem atau kustomisasi tingkat lanjut.

Manajer proses PM2 diimplementasikan untuk memastikan aplikasi berjalan secara persisten dan mampu menangani beban trafik produksi. Sebagaimana

diperlihatkan pada konfigurasi Kode 3.26, PM2 diatur dalam mode *fork* dengan fitur *autorestart* aktif. Pengaturan ini menjamin pemuatan ulang aplikasi secara otomatis apabila terjadi kegagalan sistem (*crash*) atau saat penggunaan memori melampaui ambang batas 500MB. Selain itu, manajemen log terpusat (*merge logs*) turut diaktifkan untuk mempermudah pemantauan dan audit sistem secara berkala.

```
module.exports = {
  apps: [
    {
      name: 'backend',
      script: 'dist/main.js', // Entry point aplikasi NestJS
      instances: 1,           // Mode Single Instance (Fork)
      exec_mode: 'fork',
      autorestart: true,      // Restart otomatis saat crash
      max_memory_restart: '500M', // Proteksi memori bocor (Leak)
      env_production: {
        NODE_ENV: 'production'
      },
      // Manajemen Log Terstruktur
      error_file: './logs/err.log',
      out_file: './logs/out.log',
      log_date_format: 'YYYY-MM-DD HH:mm:ss Z',
      merge_logs: true
    }
  ]
};
```

Kode 3.26: Konfigurasi PM2 Ecosystem untuk Manajemen Proses Backend

Selain itu, Nginx dikonfigurasi sebagai *Reverse Proxy* mengingat aplikasi Node.js berjalan pada gerbang internal (*port* 3000) yang tidak terekspos secara langsung. Mekanisme ini difungsikan untuk meneruskan permintaan HTTP dari jalur publik (Port 80) menuju aplikasi internal. Penerapan ini tidak hanya

menjembatani koneksi, tetapi juga memberikan lapisan keamanan tambahan pada akses server.

Pada Kode 3.27, terlihat bahwa Nginx juga dikonfigurasi untuk meneruskan informasi *header* asli klien (seperti `X-Real-IP`) ke *backend*, yang krusial untuk keperluan pencatatan jejak audit dan keamanan.

```
server {
    listen 80;
    server_name 34.101.238.244; // IP Publik Google VM

    location / {
        proxy_pass http://127.0.0.1:3000;

        # Optimasi HTTP 1.1
        proxy_http_version 1.1;
        proxy_set_header Connection "";

        # Forwarding Header Asli Klien
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
    $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Kode 3.27: Konfigurasi Nginx sebagai Reverse Proxy

B Manajemen Penyimpanan Data dan Aset (Supabase)

Lapisan basis data dipisahkan dari server aplikasi menggunakan layanan *Backend-as-a-Service* (BaaS) dari Supabase guna menjamin integritas serta skalabilitas data. Dalam arsitektur ini, basis data PostgreSQL terkelola diintegrasikan dengan aplikasi *backend* melalui ORM Prisma sebagai jembatan komunikasi data.

Batasan jumlah koneksi (*Connection Limit*) menjadi tantangan utama dalam penggunaan basis data berbasis awan (*Cloud Database*). Guna mengatasi

kendala tersebut, mekanisme *Connection Pooling* diimplementasikan melalui layanan PgBouncer yang disediakan oleh Supabase. Penerapan ini memungkinkan manajemen koneksi ke basis data menjadi lebih efisien, sehingga aplikasi tetap stabil meskipun terjadi lonjakan permintaan data secara bersamaan.

Sebagaimana terlihat pada konfigurasi `schema.prisma` di Kode 3.28, penulis memisahkan konfigurasi URL menjadi dua bagian:

- url mengarah pada *Transaction Pooler* (Port 6543) dengan parameter `pgbouncer=true`. Digunakan untuk kueri aplikasi sehari-hari guna menangani trafik tinggi.
- directUrl mengarah langsung ke sesi PostgreSQL (Port 5432). Digunakan khusus untuk operasi *Migration* yang membutuhkan koneksi persisten tanpa *pooler*.

```
// schema.prisma
datasource db {
  provider  = "postgresql"
  // Koneksi via PgBouncer (Transaction Pooler)
  url       = env("DATABASE_URL")
  // Koneksi Langsung (Session Mode) untuk Migrasi
  directUrl = env("DIRECT_DATABASE_URL")
}

// prisma.config.ts
export default defineConfig({
  schema: 'prisma/schema.prisma',
  engine: 'classic',
  datasource: {
    url: env('DATABASE_URL'),
    directUrl: env('DIRECT_DATABASE_URL'),
  },
});
```

Kode 3.28: Konfigurasi Datasource Prisma dengan Mode Connection Pooling

Selain manajemen data relasional, layanan Supabase Storage diintegrasikan pada aplikasi NestJS untuk pengelolaan aset media acara. Melalui pemanfaatan

pustaka klien (*Client Library*) Supabase, aplikasi dikonfigurasi agar dapat mengunggah berkas media secara terprogram ke dalam *bucket* bernama `event-media`. Strategi ini menjamin aset statis tersimpan secara terpusat pada penyimpanan objek (*Object Storage*) serta terdistribusi melalui CDN global, sehingga tidak membebani kapasitas penyimpanan lokal server yang terbatas.

C Integrasi Berkelanjutan Antarmuka (Vercel CI/CD)

Berbeda dengan infrastruktur *backend* yang dikelola secara imperatif pada instans virtual, distribusi antarmuka *frontend* (Next.js) menerapkan pendekatan *Serverless* melalui platform Vercel. Alur kerja *GitOps* turut diadopsi dalam proses ini, dengan menempatkan repositori Git sebagai satu-satunya sumber acuan utama (*Single Source of Truth*) untuk status infrastruktur serta kode aplikasi.

Pipeline *Continuous Deployment* (CD) dikonfigurasi untuk berjalan secara otomatis melalui tahapan berikut:

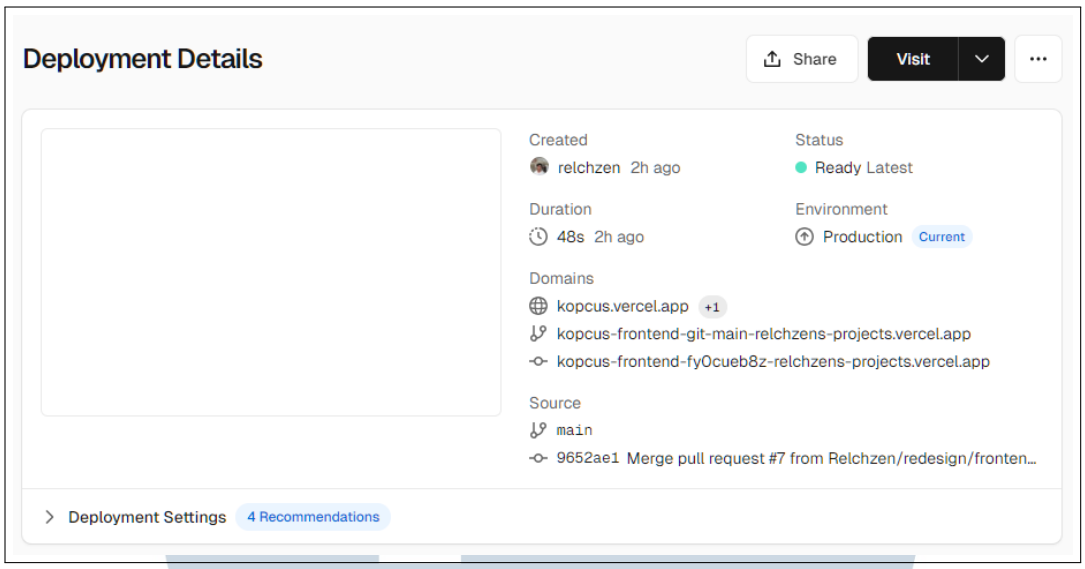
1. Deteksi Perubahan (*Change Detection*): Vercel memantau aktivitas pada cabang `main` di GitHub. Setiap operasi *push* memicu inisialisasi lingkungan *build* yang terisolasi.

2. Validasi dan Kompilasi: Sistem menjalankan perintah `next build` yang mencakup pemeriksaan tipe statis (*TypeScript Static Analysis*) dan kompilasi aset. Jika ditemukan kesalahan sintaksis atau tipe data, proses akan dihentikan secara otomatis (*Halt on Error*) untuk mencegah kode cacat mencapai lingkungan produksi.

3. Penyebaran Atomik (*Atomic Deployment*): Setelah build sukses, Vercel melakukan penyebaran versi baru ke jaringan *Edge* global. Transisi dari versi lama ke versi baru terjadi secara instan tanpa waktu henti (*Zero Downtime*).

Selain otomatisasi, keamanan konfigurasi dikelola melalui panel *Environment Variables*. Kredensial sensitif seperti `NEXT_PUBLIC_API_URL` disuntikkan ke dalam aplikasi hanya pada saat proses *build* berlangsung, sehingga tidak terekspos dalam kode sumber.

Bukti riwayat penyebaran disertakan pada Gambar 3.33 guna memvalidasi keberhasilan konfigurasi tersebut. Indikator status "*Ready*" berwarna hijau menunjukkan bahwa alur kerja (*pipeline*) CI/CD telah berjalan sukses, sehingga aplikasi dapat diakses secara publik dengan performa yang optimal.



Gambar 3.33. Riwayat *Pipeline* CI/CD pada *Dashboard* Vercel

3.3.10 Implementasi Sistem Formulir Kontak dan Notifikasi Email

Sebagai agensi digital, Kopi Chuseyo membutuhkan saluran komunikasi yang andal bagi calon klien atau mitra dalam mengirimkan permintaan penawaran (*inquiry*). Guna memenuhi kebutuhan tersebut, dikembangkanlah fitur formulir kontak yang terintegrasi langsung dengan sistem notifikasi email.

Berbeda dengan pendekatan tradisional yang mengekspos API publik secara terbuka, pola arsitektur *Backend-for-Frontend* (BFF) diterapkan menggunakan fitur *Server Actions* pada Next.js. Pemilihan pendekatan ini bertujuan untuk menyembunyikan logika komunikasi ke server *backend* utama (NestJS) sekaligus memperkuat lapisan keamanan data.

A Alur Validasi Bertingkat dan Keamanan

Mengingat formulir publik rentan terhadap serangan *spam bot*, mekanisme validasi bertingkat diimplementasikan dengan melibatkan Google reCAPTCHA v3. Alur data yang dirancang adalah sebagai berikut:

1. Sisi Klien (*Frontend*): Saat pengguna menekan tombol kirim, aplikasi meminta token validasi dari Google reCAPTCHA secara senyap (*invisible*).

2. Lapisan Middleware (Next.js *Server Action*): Token dan data formulir dikirim ke *Server Action*. Di sini, data divalidasi formatnya menggunakan pustaka

Zod untuk memastikan integritas tipe data sebelum diteruskan.

3. Sisi Server (NestJS *Backend*): *Backend* menerima data, memverifikasi validitas token reCAPTCHA ke server Google, menyimpan data ke basis data PostgreSQL melalui Prisma, dan memicu pengiriman email.

Implementasi *Server Action* yang bertindak sebagai jembatan penghubung dapat dilihat pada Kode 3.29.

```
'use server'

import { z } from 'zod';
import { contactSchema } from '@/lib/schemas';

export async function submitContactInquiry(prevState:
    any, formData: FormData) {
  // 1. Validasi Input menggunakan Zod
  const validatedFields = contactSchema.safeParse({
    name: formData.get('name'),
    email: formData.get('email'),
    // ... field lainnya
    captchaToken: formData.get('g-recaptcha-response'),
  });

  if (!validatedFields.success) {
    return { success: false, errors: validatedFields.
    error.flatten().fieldErrors };
  }

  // 2. Teruskan ke Backend NestJS (Internal Network)
  try {
    const response = await fetch(`${process.env.
    CHUSEYO_API_URL}/contact/submit`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(validatedFields.data),
    });
```

```
26
27      if (!response.ok) throw new Error('Backend
    rejection');
28      return { success: true, message: 'Permintaan
    berhasil dikirim.' };
29
30    } catch (error) {
31      return { success: false, message: 'Gagal
    menghubungi server.' };
32    }
33  }
```

Kode 3.29: Implementasi Server Action sebagai BFF untuk Formulir Kontak

B Integrasi Layanan Pengiriman Surel (Resend)

Layanan Resend digunakan untuk menangani pengiriman surel transaksional. Infrastruktur ini dipilih karena menyediakan API berbasis HTTP yang lebih modern dan cepat dibandingkan protokol SMTP konvensional, serta memiliki tingkat keberhasilan pengiriman (*deliverability*) yang tinggi.

Pada sisi *backend* (NestJS), logika bisnis diatur untuk melakukan dua aksi atomik: menyimpan data formulir sebagai arsip di basis data, dan mengirimkan notifikasi ke email admin perusahaan. Implementasi logika layanan tersebut ditunjukkan pada Kode 3.30.

```
1  async handleSubmission(dto: ContactSubmissionDto, meta:
       { ip: string }) {
2      // 1. Verifikasi Token Captcha ke Google
3      const isHuman = await this.captchaService.
       validateToken(dto.captchaToken);
4      if (!isHuman) throw new ForbiddenException('
       Aktivitas mencurigakan terdeteksi');
5
6      // 2. Simpan Arsip ke Database (Prisma)
7      const submission = await this.prisma.
       contactSubmission.create({
8          data: {
```

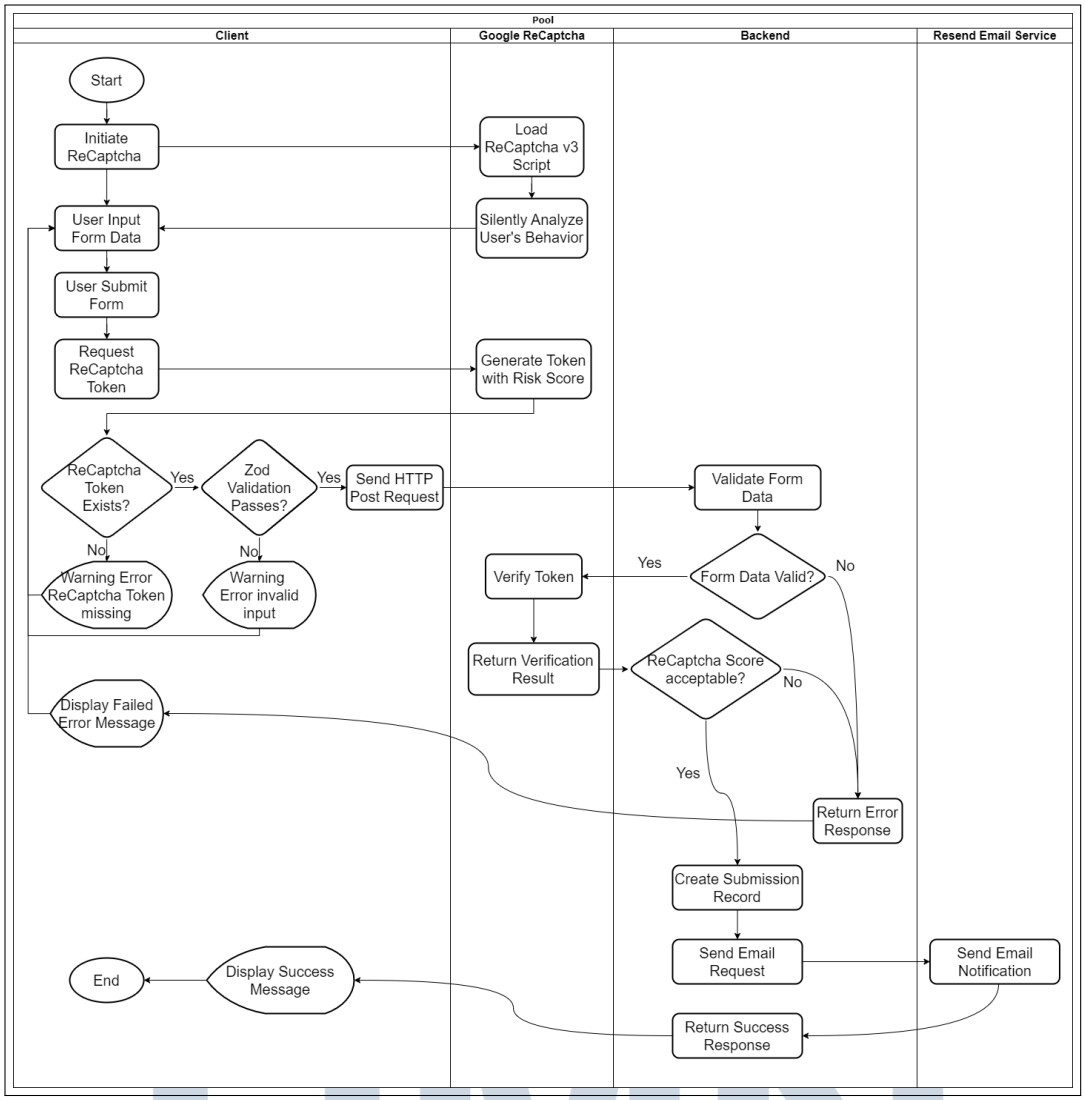
```
            fullName: dto.name,
            email: dto.email,
            companyName: dto.company,
            projectType: dto.projectType,
            ipAddress: meta.ip,
        }
    });

    // 3. Kirim Notifikasi Email via Resend
    await this.emailService.sendNotification({
        to: process.env.ADMIN_EMAIL,
        subject: `New Inquiry: ${dto.company}`,
        html: `<p>Permintaan baru dari <b>${dto.name}</b>...</p>`
    });

    return { success: true };
}
```

Kode 3.30: Layanan *Backend* untuk Penyimpanan Data dan Notifikasi Email

Untuk memperjelas mekanisme keamanan yang diterapkan, Gambar 3.34 mengilustrasikan perjalanan data dari sisi pengguna hingga notifikasi diterima oleh administrator. Diagram ini menunjukkan pemisahan lapisan publik dan privat untuk mencegah kebocoran kredensial serta memitigasi serangan *spam*.

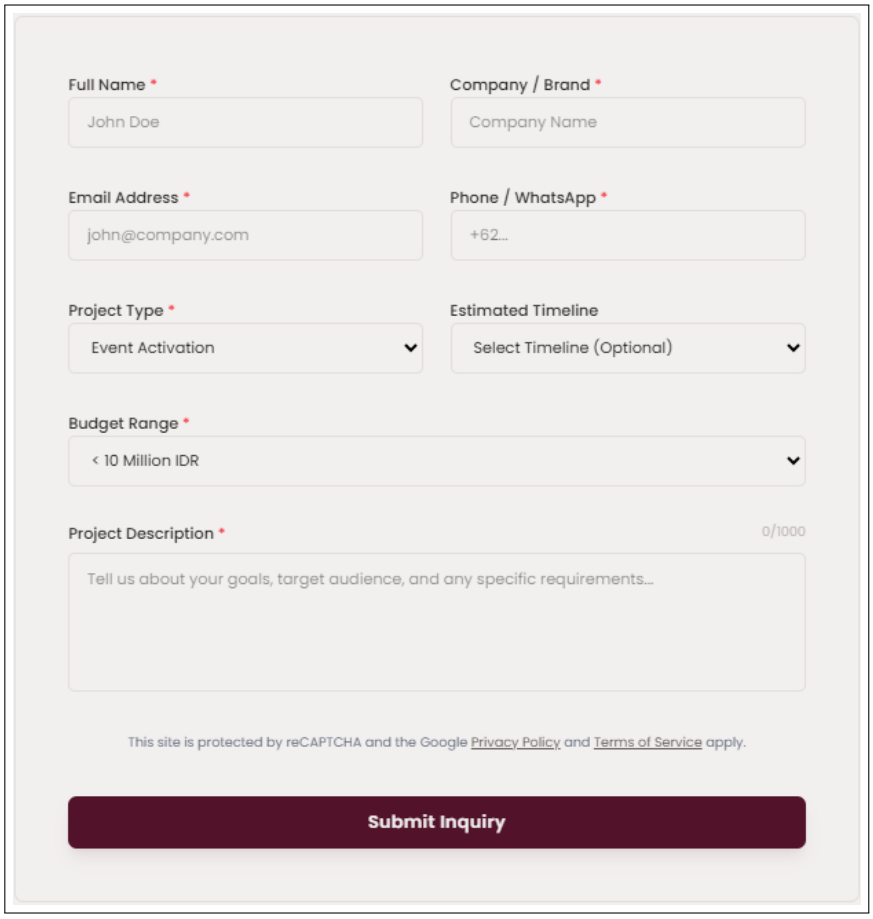
Berdasarkan alur diagram, proses dimulai pada sisi *Client* dengan inisiasi skrip *Google ReCaptcha v3* yang melakukan analisis perilaku pengguna secara senyap (*silently analyze*). Sebelum data dikirim ke server, sistem menerapkan validasi awal untuk memastikan token keamanan telah terbuat dan format input sesuai dengan skema *Zod Validation*.

Apabila validasi sisi klien terpenuhi, data dikirim melalui *HTTP POST Request* menuju lapisan *Backend*. Di lapisan ini, validasi ulang dilakukan secara ketat, diikuti dengan proses verifikasi token (*Verify Token*) ke server Google untuk menilai skor risiko pengguna. Jika skor dinyatakan aman (*acceptable*), sistem akan menyimpan catatan submisi ke basis data dan meneruskan permintaan pengiriman notifikasi melalui layanan eksternal, *Resend Email Service*, sebelum akhirnya mengirimkan umpan balik kesuksesan (*Success Response*) kembali ke pengguna.

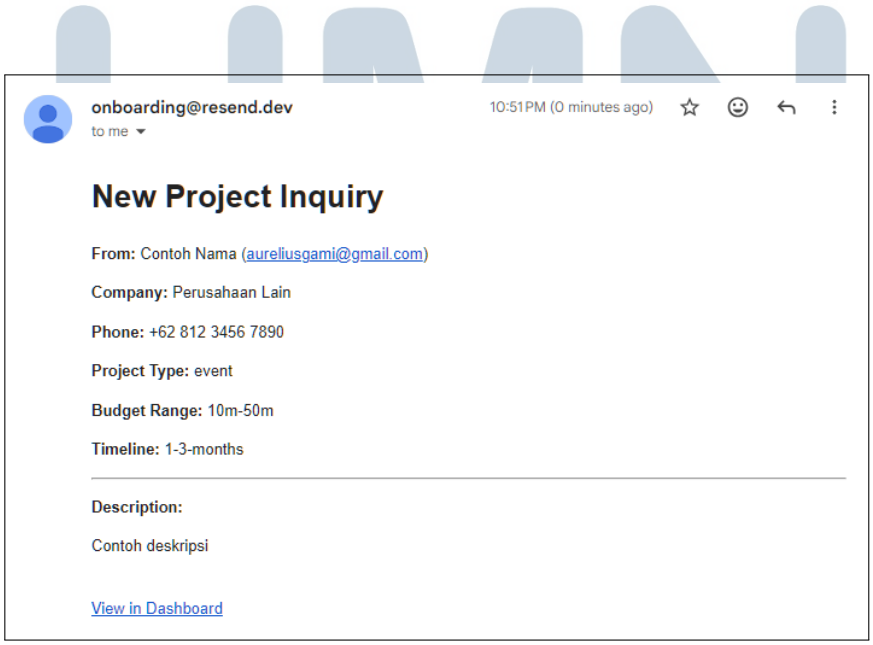


Gambar 3.34. Diagram Swimlane Alur Validasi Token reCAPTCHA dan Distribusi Email

Validasi fungsional dilakukan secara menyeluruh untuk memastikan integrasi antarmuka dan layanan pengiriman pesan berjalan akurat. Implementasi formulir kontak yang telah dilengkapi mekanisme validasi input visual ditunjukkan pada Gambar 3.35, sedangkan bukti keberhasilan alur komunikasi data ditampilkan pada Gambar 3.36, di mana notifikasi surel berhasil diterima administrator secara *real-time* melalui layanan *Resend* lengkap dengan rincian data pengguna.



Gambar 3.35. Antarmuka Formulir Kontak pada Situs Web Kopi Chuseyo



Gambar 3.36. Bukti Notifikasi Surel yang Diterima oleh Administrator via Layanan *Resend*

3.4 Kendala dan Solusi yang Ditemukan

Selama fase pengembangan dan implementasi sistem, terdapat beberapa hambatan teknis yang muncul, terutama terkait dengan efisiensi infrastruktur serta keterbatasan sumber daya virtual.

3.4.1 Kendala yang Ditemukan

Dalam proses *deployment* dan pemeliharaan sistem, ditemukan beberapa persoalan utama yang menjadi hambatan, antara lain:

1. Inefisiensi Pemanfaatan Sumber Daya
   Pada fase awal, penggunaan dua unit *Virtual Machine* (VM) tipe *e2-medium* dengan kapasitas RAM 4GB untuk memisahkan layanan *backend* dan *CMS* dinilai tidak optimal. Berdasarkan data pemantauan, rata-rata konsumsi sumber daya berada di bawah 20%, sehingga alokasi spesifikasi tersebut memicu pemborosan biaya sewa server bulanan.

2. Instabilitas pada Spesifikasi Minimal
   Upaya penghematan biaya dengan menurunkan spesifikasi server ke tipe *e2-small* (RAM 2GB) menghadapi kendala teknis serius. Kapasitas memori yang terbatas menyebabkan server sering mengalami kegagalan (*crash*) saat dipaksa menjalankan proses kompilasi (*build*) aplikasi Strapi dan NestJS yang cukup berat secara langsung di sisi server.

3. Tantangan Manajemen Lalu Lintas Data
   Penyatuan dua layanan yang berbeda ke dalam satu unit server fisik menimbulkan kompleksitas pada sisi jaringan. Dibutuhkan mekanisme yang mampu memilah dan mengarahkan permintaan dari domain `api` dan `cms` ke *port* internal masing-masing aplikasi secara tepat melalui satu alamat IP publik.

3.4.2 Solusi yang Ditemukan

Guna mengatasi berbagai hambatan tersebut, telah diimplementasikan beberapa solusi teknis yang komprehensif sebagai berikut:

1. Penyederhanaan Infrastruktur
   Melakukan penggabungan (*merging*) kedua layanan utama ke dalam satu unit

server *e2-small*. Langkah ini diambil untuk memaksimalkan utilitas sumber daya yang ada sekaligus memangkas pengeluaran operasional infrastruktur cloud secara signifikan.

2. Optimasi Alur Kerja dan Manajemen Memori
   Untuk mengatasi keterbatasan RAM, diadopsi mekanisme *off-server build* dengan memindahkan seluruh proses kompilasi ke jalur *CI/CD* GitHub Actions. Dengan cara ini, server hanya bertugas menjalankan aplikasi yang sudah siap pakai. Selain itu, digunakan *process manager* PM2 untuk menjaga efisiensi proses serta penambahan *Swap Memory* sebesar 2GB sebagai jaring pengaman apabila terjadi lonjakan pemakaian memori.

3. Implementasi *Reverse Proxy* melalui Nginx
   Untuk menjamin kemudahan akses, Nginx dikonfigurasi sebagai *reverse proxy* yang berfungsi sebagai gerbang tunggal lalu lintas data. Nginx secara otomatis memetakan permintaan dari domain `api.kopichuseyo.com` dan `cms.kopichuseyo.com` menuju porta internal yang relevan, sehingga kedua layanan dapat beroperasi berdampingan tanpa konflik akses.