Dokumentasi Komprehensif: Perbaikan Sistem CRUD Locker

Versi Dokumen: 1.0
Tanggal: 2025-06-09

1. Pendahuluan

Dokumentasi ini memberikan panduan teknis lengkap mengenai pembaruan terkini pada sistem manajemen loker, khususnya pada layanan `databaseService`. Perbaikan ini mencakup penyempurnaan operasi **CRUD (Create, Read, Update, Delete)**, pengenalan sistem manajemen ketersediaan loker otomatis, dan peningkatan logika bisnis secara keseluruhan.

Tujuan dari pembaruan ini adalah untuk menciptakan sistem yang lebih **robust, andal, dan mudah dikelola** oleh developer. Dokumentasi ini ditujukan bagi para developer yang akan mengintegrasikan, menggunakan, atau memelihara sistem ini.

---

2. Ringkasan Perbaikan

Berikut adalah poin-poin utama perbaikan yang telah diimplementasikan:

A. Kelengkapan Operasi CRUD

- **ESP32 Devices**: Operasi `DELETE` yang sebelumnya tidak ada kini telah ditambahkan.
- **Payments**: Operasi `CREATE` yang vital untuk memulai transaksi kini telah tersedia.
- **Locker Logs**: Operasi `CREATE`, `UPDATE`, dan `DELETE` telah ditambahkan untuk pelacakan audit yang lengkap.

B. Locker Availability Management System

- **Otomatisasi Status Loker**: Implementasi `class LockerAvailabilityManager` untuk mengelola status loker (`available` atau `occupied`) secara otomatis berdasarkan siklus hidup transaksi.
- **Pencatatan Otomatis**: Setiap perubahan status loker kini dicatat secara otomatis dalam `Locker Logs`, memastikan jejak audit yang transparan.
- **Manajemen Jumlah Ketersediaan**: Jumlah loker yang tersedia diperbarui secara real-time tanpa intervensi manual.

C. Peningkatan Fitur dan Logika Bisnis

- **Penanganan Error**: Pesan error kini lebih spesifik dan informatif, membantu proses debugging menjadi lebih cepat.
- **Validasi Data**: Peningkatan pada proses transformasi dan validasi data untuk memastikan integritas.
- **Operasi Aman (Transaction-Safe)**: Operasi kritis kini bersifat transaction-safe dengan kemampuan rollback untuk mencegah inkonsistensi data.
- **Sinkronisasi Real-Time**: Sinkronisasi data dengan Firebase Realtime Database telah ditingkatkan untuk performa yang lebih baik.

---

3. Panduan Penggunaan API (`databaseService.ts`)

Layanan `databaseService` adalah single source of truth untuk semua interaksi dengan database. Berikut adalah panduan penggunaan untuk setiap entitas.

Entitas: `lockers`

- **Tujuan**: Mengelola data master loker.

- **Operasi**:
  - `getLockers()`: Mengambil semua data loker.
  - `getLockerById(id: string)`: Mengambil loker spesifik berdasarkan ID.
  - `createLocker(data: Locker)`: Membuat loker baru.
  - `updateLocker(id: string, updates: Partial<Locker>)`: Memperbarui data loker.
  - `deleteLocker(id: string)`: Menghapus data loker.

Entitas: `locker-logs`

- **Tujuan**: Mencatat semua aktivitas dan perubahan status pada loker.
- **Operasi**:
  - `getLockerLogs()`: Mengambil semua log.
  - `getLockerLogsByLockerId(lockerId: string)`: Mengambil log untuk loker tertentu.
  - `createLockerLog(data: LockerLog)`: **(Baru)** Membuat catatan log baru. Umumnya dipanggil secara otomatis oleh sistem.
  - `updateLockerLog(id: string, updates: Partial<LockerLog>)`: **(Baru)** Memperbarui log jika diperlukan.
  - `deleteLockerLog(id: string)`: **(Baru)** Menghapus data log.

Entitas: `esp32-devices`

- **Tujuan**: Mengelola perangkat keras ESP32 yang terhubung ke setiap loker.

- **Operasi**:
  - `getEsp32Devices()`: Mengambil semua data perangkat.
  - `getEsp32DeviceById(id: string)`: Mengambil perangkat spesifik.
  - `createEsp32Device(data: Esp32Device)`: Mendaftarkan perangkat baru.
  - `updateEsp32Device(id: string, updates: Partial<Esp32Device>)`: Memperbarui data perangkat.
  - `deleteEsp32Device(id: string)`: **(Baru)** Menghapus perangkat dari sistem.

Entitas: `transactions`

- **Tujuan**: Mengelola transaksi penyewaan loker.
- **Operasi**:
  - `getTransactions()`: Mengambil semua data transaksi.
  - `getTransactionById(id: string)`: Mengambil transaksi spesifik.
  - `createTransaction(data: Transaction)`: Membuat transaksi baru.
  - `updateTransaction(id: string, updates: Partial<Transaction>)`: Memperbarui status atau detail transaksi.
  - `deleteTransaction(id: string)`: Menghapus data transaksi.

Entitas: `payments`

- **Tujuan**: Mengelola data pembayaran yang terkait dengan transaksi.

- **Operasi**:
  - `getPayments()`: Mengambil semua data pembayaran.
  - `getPaymentByTransactionId(transactionId: string)`: Mengambil pembayaran untuk transaksi tertentu.
  - `createPayment(data: Payment)`: **(Baru)** Membuat entri pembayaran baru saat transaksi dimulai.
  - `updatePayment(id: string, updates: Partial<Payment>)`: Memperbarui status pembayaran (misalnya, dari `pending` ke `success`).
  - `deletePayment(id: string)`: Menghapus data pembayaran.

---

4. Cara Kerja Locker Availability Management System

Sistem ini dirancang untuk mengotomatiskan tugas paling kritis dan rentan kesalahan: mengelola ketersediaan loker.

Komponen Utama: `LockerAvailabilityManager`

`LockerAvailabilityManager` adalah sebuah class yang berjalan di background dan terintegrasi penuh dengan siklus hidup transaksi.

Alur Kerja Otomatis:

1. **Inisiasi Transaksi**: Saat `createTransaction` berhasil dipanggil untuk sebuah loker, `LockerAvailabilityManager` akan "mendengarkan" event ini.
2. **Perubahan Status**: Manajer secara otomatis memanggil `updateLocker` untuk mengubah status loker dari `available` menjadi `occupied`.
3. **Pembuatan Log**: Secara bersamaan, manajer memanggil `createLockerLog` untuk mencatat aktivitas ini, misalnya: `Locker A-01 status changed to occupied due to transaction TXN-123`.

4. **Rilis Loker**: Ketika pengguna mengambil barangnya (melalui `lockerRetrievalService`) atau jika pembayaran gagal (`payment failure`), manajer akan kembali bertindak.
5. **Status Kembali Tersedia**: Status loker yang bersangkutan diubah kembali menjadi `available`.
6. **Log Rilis**: Log baru dibuat untuk mencatat pelepasan loker, misalnya: `Locker A-01 status changed to available after item retrieval for transaction TXN-123`.

Dengan sistem ini, developer tidak perlu lagi mengelola status loker secara manual, sehingga mengurangi risiko human error dan memastikan data selalu konsisten.

---

5. Contoh Implementasi CRUD

Berikut adalah contoh pseudo-code dalam TypeScript untuk menunjukkan cara menggunakan `databaseService`.

Contoh 1: Membuat Transaksi dan Pembayaran Baru

```
import { databaseService } from './services/databaseService';

async function startNewLockerRental(userId: string, lockerId:
string) {
  try {
    // 1. Membuat transaksi baru
    const transactionData = { userId, lockerId, startTime: new
Date(), status: 'pending' };
    const newTransaction = await
databaseService.createTransaction(transactionData);
    console.log('Transaction created:', newTransaction);

    // 2. (BARU) Membuat entri pembayaran terkait
    const paymentData = {
      transactionId: newTransaction.id,
      amount: 5000, // Contoh biaya sewa
      status: 'unpaid'
    };
    const newPayment = await
databaseService.createPayment(paymentData);
    console.log('Payment entry created:', newPayment);

    // Pada titik ini, LockerAvailabilityManager akan otomatis
mengubah status loker menjadi 'occupied'
    // dan membuat log terkait.

    return { transaction: newTransaction, payment: newPayment };

  } catch (error) {
    console.error('Failed to start rental:', error.message);
    // Logika rollback akan dijalankan oleh service jika terjadi
kegagalan
  }
}
```

Contoh 2: Menghapus Perangkat ESP32

```
import { databaseService } from './services/databaseService';

async function decommissionDevice(deviceId: string) {
  try {
    // (BARU) Memanggil fungsi deleteEsp32Device
    await databaseService.deleteEsp32Device(deviceId);
    console.log(`Device with ID ${deviceId} has been successfully
decommissioned.`);
  } catch (error) {
    console.error(`Failed to delete device ${deviceId}:`,
error.message);
  }
}
```

Contoh 3: Membaca Log Aktivitas Loker

```
import { databaseService } from './services/databaseService';

async function getLockerHistory(lockerId: string) {
  try {
    const logs = await
databaseService.getLockerLogsByLockerId(lockerId);
    if (logs.length > 0) {
      console.log(`Activity logs for Locker ${lockerId}:`);
      logs.forEach(log => {
        console.log(`- [<span class="math-inline" style="display:
inline;"><math xmlns="http://www.w3.org/1998/Math/MathML"
display="inline"><mrow><mrow><mi>n</mi><mi>e</mi><mi>w</mi><mi>D</
mi><mi>a</mi><mi>t</mi><mi>e</mi><mo stretchy="false">&#x00028;</
mo><mi>l</mi><mi>o</mi><mi>g</mi><mo>&#x0002E;</mo><mi>t</
mi><mi>i</mi><mi>m</mi><mi>e</mi><mi>s</mi><mi>t</mi><mi>a</
mi><mi>m</mi><mi>p</mi><mo stretchy="false">&#x00029;</
mo><mo>&#x0002E;</mo><mi>t</mi><mi>o</mi><mi>L</mi><mi>o</
mi><mi>c</mi><mi>a</mi><mi>l</mi><mi>e</mi><mi>S</mi><mi>t</
mi><mi>r</mi><mi>i</mi><mi>n</mi><mi>g</mi><mo
stretchy="false">&#x00028;</mo><mo stretchy="false">&#x00029;</
mo></mrow><mo stretchy="false">]</mo></mrow></math></
span>{log.message}`);
      });
    } else {
      console.log(`No activity logs found for Locker $
{lockerId}.`);
    }
  } catch (error) {
    console.error('Failed to retrieve locker logs:',
error.message);
  }
}
```

---

6. Best Practices

Untuk memastikan sistem berjalan optimal dan andal, ikuti praktik terbaik berikut:

1. **Jangan Ubah Status Loker Secara Manual**: Selalu andalkan `LockerAvailabilityManager`. Memperbarui status loker secara manual akan mengganggu alur kerja otomatis dan dapat menyebabkan inkonsistensi data.
2. **Gunakan Penanganan Error yang Tepat**: Selalu bungkus pemanggilan fungsi `databaseService` dalam blok `try...catch` untuk menangani potensi kegagalan, seperti masalah jaringan atau validasi.
3. **Validasi Data di Sisi Klien**: Sebelum mengirim data ke API (misalnya saat membuat transaksi), pastikan semua data yang diperlukan sudah valid untuk mengurangi panggilan API yang gagal.
4. **Manfaatkan Log untuk Debugging**: Jika terjadi perilaku yang tidak terduga pada sebuah loker, periksa `locker-logs` terlebih dahulu. Ini adalah sumber informasi pertama untuk melacak apa yang terjadi.
5. **Jaga Konsistensi ID**: Pastikan untuk menggunakan ID yang benar saat menghubungkan entitas, seperti `transactionId` di dalam objek `Payment`.

---

7. Panduan Migrasi dari Versi Sebelumnya

Karena perbaikan ini menjaga kompatibilitas mundur (backward compatibility), proses migrasi relatif sederhana.

Langkah 1: Tinjau Kode yang Ada
Identifikasi semua bagian kode Anda yang berinteraksi dengan `databaseService`. Beri perhatian khusus pada area yang sebelumnya "mem-bypass" fungsionalitas yang hilang.

Langkah 2: Hapus Logika Manual untuk Status Loker
Cari dan hapus kode apa pun yang secara manual mengubah status `lockers`. Misalnya, jika Anda sebelumnya memiliki kode seperti:

```
// Kode lama yang harus dihapus
updateLocker(lockerId, { status: 'occupied' });
```

Kode ini tidak lagi diperlukan karena sistem baru menanganinya secara otomatis.

Langkah 3: Implementasikan Pembuatan Pembayaran
Pada alur kerja pembuatan transaksi, tambahkan pemanggilan `createPayment` setelah transaksi berhasil dibuat, seperti yang ditunjukkan pada Contoh 1 di atas.

Langkah 4: Manfaatkan Fungsionalitas `DELETE` Baru
Jika aplikasi Anda memiliki fitur untuk menghapus perangkat, sekarang Anda dapat mengimplementasikannya dengan memanggil `deleteEsp32Device(id)`.

Langkah 5: Pengujian Menyeluruh
Jalankan semua alur kerja utama setelah migrasi:
- Membuat transaksi baru.
- Membatalkan transaksi.
- Menyelesaikan penyewaan (pengambilan barang).
- Memastikan status loker dan log diperbarui secara otomatis di setiap langkah.
- Gunakan skrip `test_enhanced_database_service.py` sebagai referensi untuk pengujian Anda.

Dengan mengikuti panduan ini, migrasi ke sistem yang telah diperbaiki dapat berjalan lancar tanpa mengganggu fungsionalitas yang sudah ada.