BAB IV
PEMBUATAN ALAT

4.1 Pembuatan Perangkat Keras

Pembuatan perangkat keras merupakan tahapan awal dalam proses perancangan sistem dispenser *earplug* otomatis berbasis RFID dan ESP32. Pada tahap ini, dilakukan perakitan seluruh komponen fisik yang mendukung kinerja alat baik dari sisi kelistrikan maupun mekanikal. Perangkat keras yang dirancang terdiri dari beberapa komponen utama, yaitu mikrokontroler ESP32 sebagai pusat pengendali sistem, modul RFID untuk identifikasi pengguna, servo motor sebagai aktuator mekanik untuk mengeluarkan earplug, sensor TCRT5000 untuk mendeteksi keluarnya earplug, OLED display sebagai antarmuka visual, serta modul LM2596 untuk pengaturan tegangan. Selain sebagai pusat pengendali sistem, ESP32 juga berfungsi untuk bertukar data dengan *database* melalui jaringan internet. *Database* berperan sebagai *data logger* yang akan terhubung pada Website dan aplikasi *mobile* untuk menampilkan data.

4.1.1 Alat dan Bahan

Proses pertama dalam pembuatan sistem ini adalah mempersiapkan alat dan bahan yang akan digunakan. Daftar alat yang digunakan untuk membangun sistem dispenser *earplug* otomatis berbasis RFID dan ESP32 dengan monitoring melalui website dan aplikasi *mobile* dapat dilihat pada Tabel 4-1.

Tabel 4-1 Alat yang digunakan

No	Nama Alat	Jumlah
1	Multimeter	1 buah
2	Obeng Set	1 buah
3	Gunting	1 buah
4	Tang potong	1 buah
5	Tang kupas	1 buah

No	Nama Alat	Jumlah
6	Solder	1 buah
7	Bor tangan set	1 buah
8	Jangka Sorong	1 buah
9	Alat tulis	1 set
10	3D Print	1 buah

Pada Tabel 4-2 merupakan bahan yang digunakan untuk membangun sistem dispenser *earplug* otomatis berbasis RFID dan ESP32 dengan monitoring melalui website dan aplikasi *mobile*.

Tabel 4-2 Bahan yang digunakan

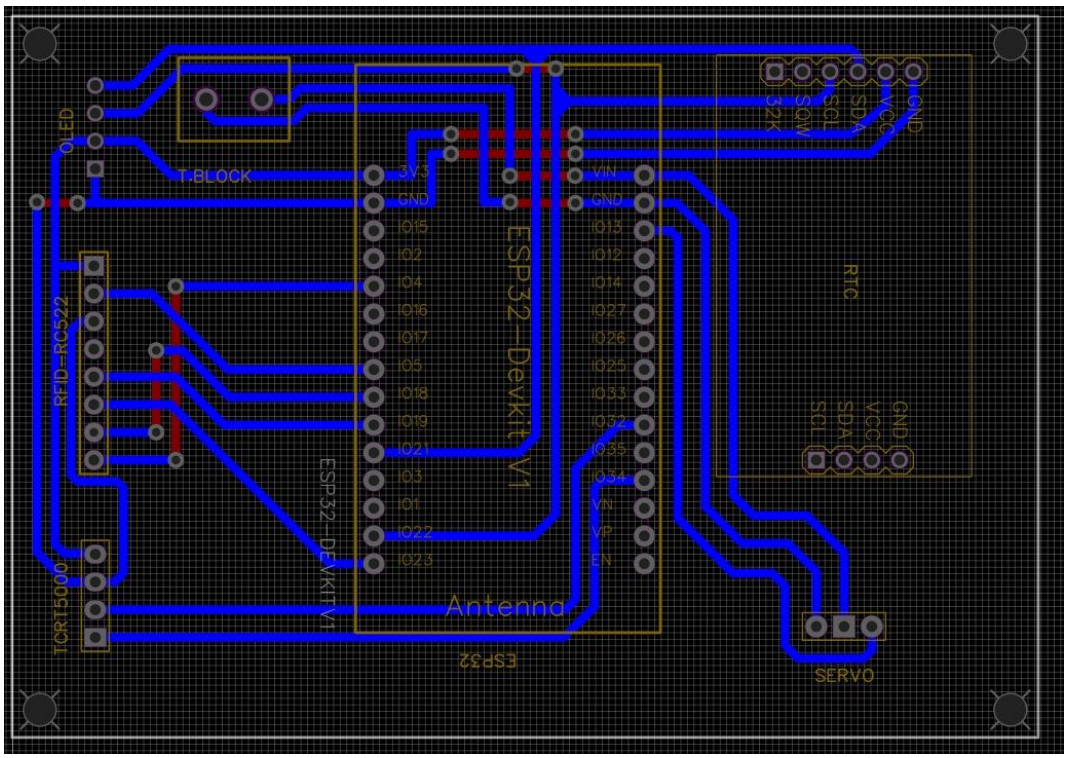
No	Nama Bahan	Jumlah
1	ESP32 DEVKIT V1	1 buah
2	Modul RFID RC-522	1 buah
3	Kartu Tag RFID	4 buah
4	OLED Display 64x128	1 buah
5	Servo MG996R	1 buah
6	Modul TCRT5000	1 buah
7	Modul LM2596	1 buah
8	Power Supply 12V 3A	1 buah
9	Kabel data USB to Micro B	1 buah
10	PCB	1 buah
11	Timah Solder	1 Rol
12	Filamen 3D print jenis PLA+	2 kg

Seluruh alat dan bahan yang digunakan dipilih berdasarkan ketersediaan, keandalan, serta kesesuaian dengan kebutuhan sistem. Penggunaan komponen modular seperti ESP32 Devkit V1, RFID RC-522, LM2596, dan TCRT5000 digunakan untuk memudahkan proses integrasi dan pemeliharaan sistem.

4.1.2 Pembuatan Perangkat Elektrik

Pembuatan perangkat elektrik merupakan tahap penting dalam proses perakitan sistem dispenser *earplug* otomatis. Pada tahap ini, dilakukan perancangan dan perakitan seluruh komponen elektronik yang bertugas mengendalikan sistem, membaca RFID, menggerakkan servo, dan mendeteksi keberhasilan pengeluaran *earplug*. Seluruh komponen elektronik ini dirangkai dalam satu sistem berbasis mikrokontroler ESP32. Adapun langkah-langkah pembuatan perangkat elektrik adalah sebagai berikut.

1. Perancangan Skematik dan Layout PCB



Gambar 4.1 Layout PCB

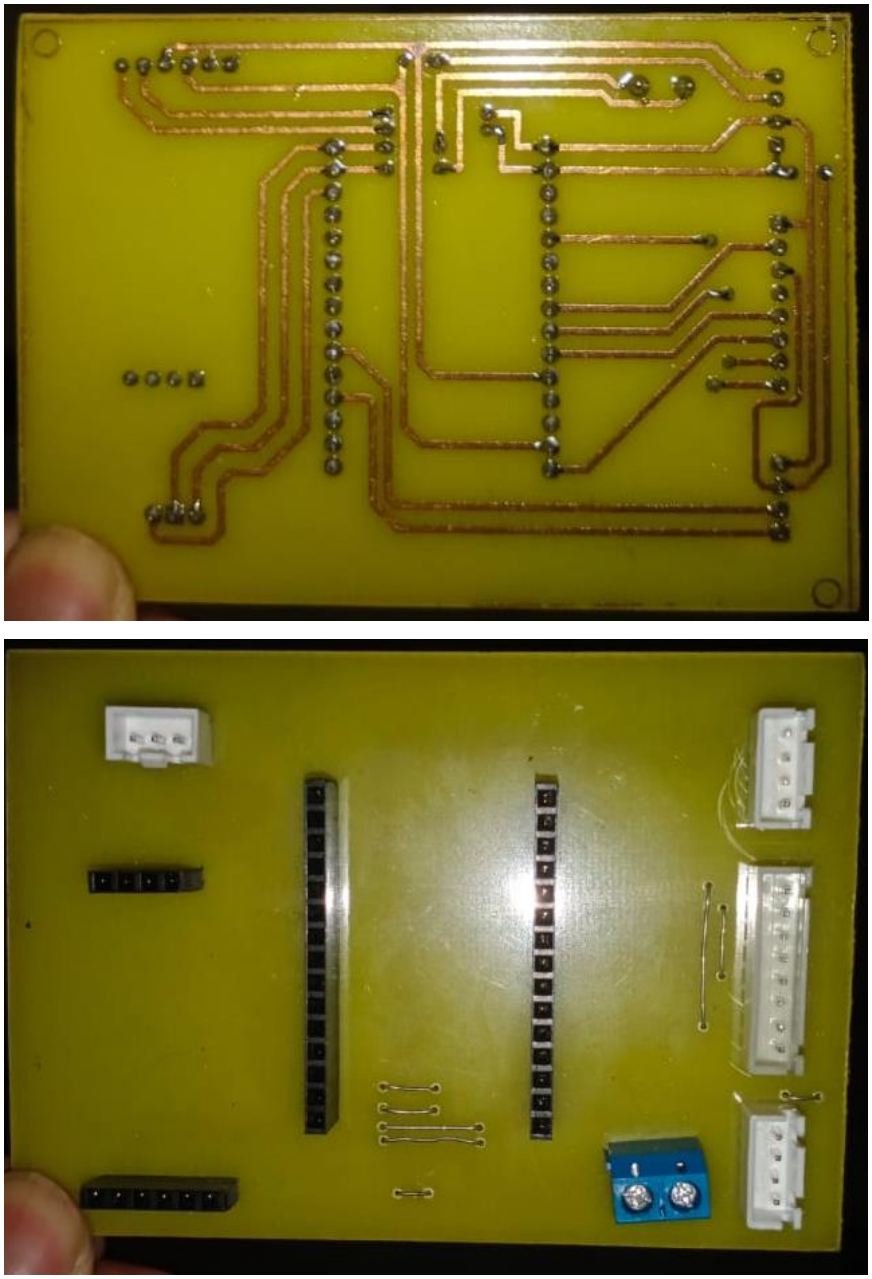
Langkah awal dimulai dengan membuat skematik rangkaian elektronik menggunakan perangkat lunak EasyEDA seperti pada Gambar 4.1. Dalam skematik ini, seluruh komponen seperti ESP32, modul RFID RC522, OLED Display, sensor TCRT5000, servo motor, dan modul step-down LM2596 disusun secara terstruktur agar koneksi antar komponen dapat ditentukan dengan jelas. Setelah skematik

selesai, dilakukan perancangan layout PCB berdasarkan koneksi pada skematik tersebut. Proses ini mencakup penempatan jalur (*routing*), pengaturan ukuran papan, serta pemberian label dan lubang untuk konektor JST yang akan digunakan untuk menghubungkan komponen eksternal.

2. Proses Pencetakan dan Penyolderan PCB

Setelah layout selesai, file desain dicetak menggunakan printer laser. Berikutnya dilakukan proses transfer jalur ke papan tembaga. Kemudian *Etching* dilakukan dengan merendam papan ke dalam larutan *ferric chloride* ($FeCl_3$) untuk melarutkan bagian tembaga yang tidak terlindungi jalur. Setelah proses *etching* selesai, jalur yang terbentuk dibersihkan, lalu dilubangi menggunakan bor PCB untuk pemasangan komponen. Komponen yang digunakan tidak disolder langsung ke PCB melainkan akan dipasang pin header dan pin JST. ESP32 akan dipasang ke PCB melalui pin *header*. Sedangkan OLED, modul RFID RC-522, modul TCRT5000, dan servo dihubungkan ke PCB melalui pin JST. Dengan menggunakan pin JST, komponen dapat diletakkan secara fleksibel agar dapat menyesuaikan kebutuhan sistem. Pada Gambar 4.2 ditunjukkan proses penyolderan komponen ke PCB, sedangkan Gambar 4.3 menunjukkan PCB yang telah dicetak dan disolder.

Gambar 4.2 Proses Penyolderan PCB

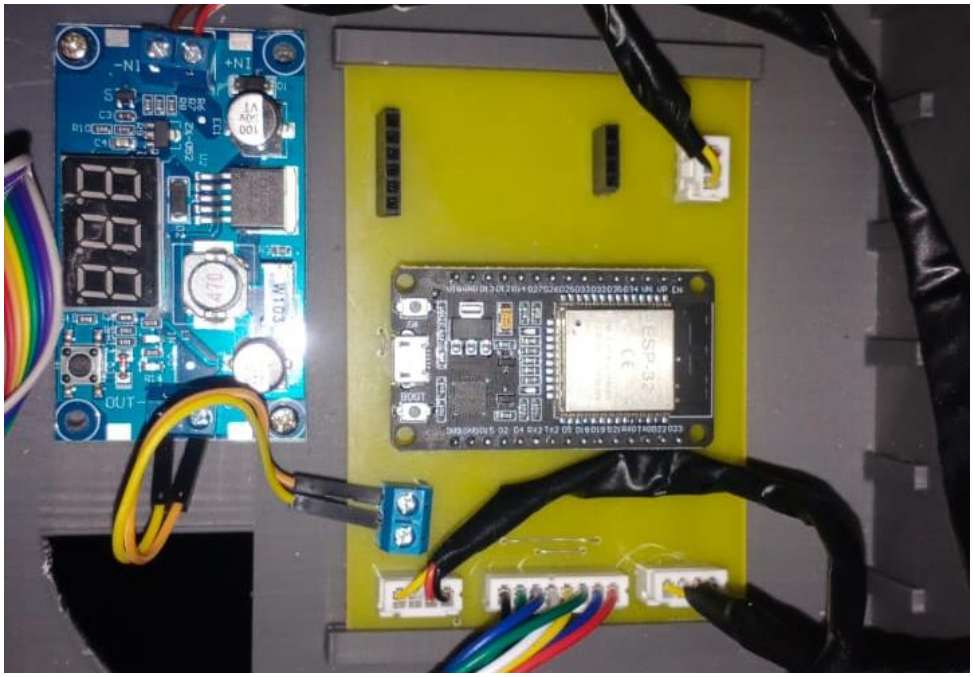


Gambar 4.3 Hasil Pencetakan dan Penyolderan PCB

3. Integrasi Sistem

Beberapa komponen seperti OLED display, servo motor, sensor TCRT5000, dan modul RFID RC522 tidak langsung disolder ke PCB, tetapi dipasang menggunakan kabel JST. Tujuan pemasangan ini adalah untuk memberikan fleksibilitas posisi saat integrasi dengan case 3D print serta memudahkan proses perawatan dan

penggantian komponen apabila diperlukan. Gambar 4.4 menunjukkan komponen yang sudah terpasang di PCB.



Gambar 4.4 Komponen yang Sudah Terpasang di PCB

4.1.3 Pembuatan Perangkat Mekanik

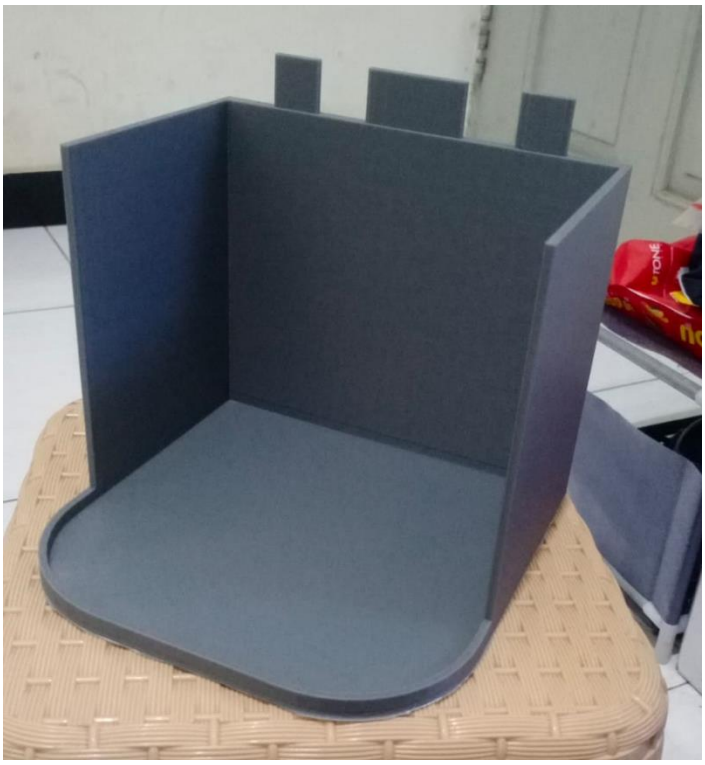
Perangkat mekanik pada sistem dispenser earplug otomatis dirancang untuk mendukung proses pengeluaran earplug secara otomatis dan terintegrasi dengan komponen elektronik. Seluruh bagian mekanik, mulai dari wadah penyimpanan, mekanisme pengeluaran, hingga dudukan komponen elektronik, dibuat menggunakan teknologi pencetakan tiga dimensi (*3D printing*) dengan material filamen PLA+. Adapun langkah-langkah pembuatan perangkat mekanik adalah sebagai berikut.

1. Desain Komponen Mekanik

Desain komponen dilakukan secara menyeluruh menggunakan perangkat lunak Autodesk Inventor. Sistem mekanik pengeluaran *earplug* dirancang menggunakan mekanisme rotasi yang digerakkan oleh servo motor, sehingga *earplug* dapat dikeluarkan secara otomatis dalam jumlah yang tepat. Selain itu, rancangan casing juga mencakup ruang untuk sensor TCRT5000 guna mendeteksi jatuhnya earplug,

serta ruang sirkulasi untuk menjaga kestabilan suhu dan kelembaban di dalam dispenser. Secara keseluruhan, sistem ini terdiri dari sembilan komponen mekanik utama yang seluruhnya dicetak menggunakan teknologi 3D printing dengan material PLA+. Komponen-komponen tersebut meliputi:

1. Penyangga bawah yang ditunjukkan oleh Gambar 4.5, berfungsi sebagai fondasi alat yang menopang seluruh struktur.



Gambar 4.5 Penyangga Bawah

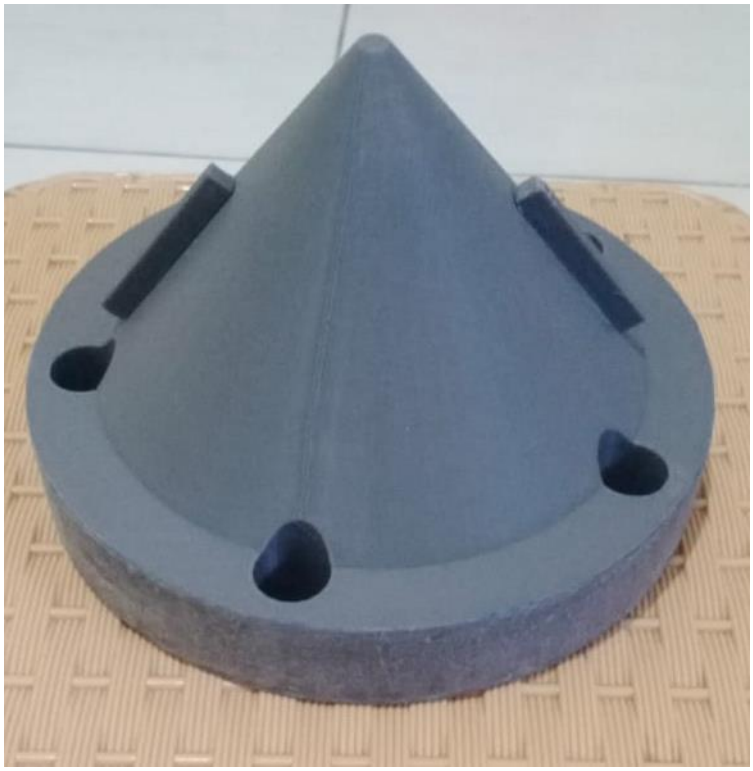
2. *Casing* tengah bawah yang ditunjukkan oleh Gambar 4.6, sebagai tempat penempatan sebagian besar komponen elektronik dan mekanik.

Gambar 4.6 Casing Tengah

3. *Center plate* bawah yang ditunjukkan oleh Gambar 4.7, sebagai penghubung struktural antara bagian bawah dan bagian atas alat.

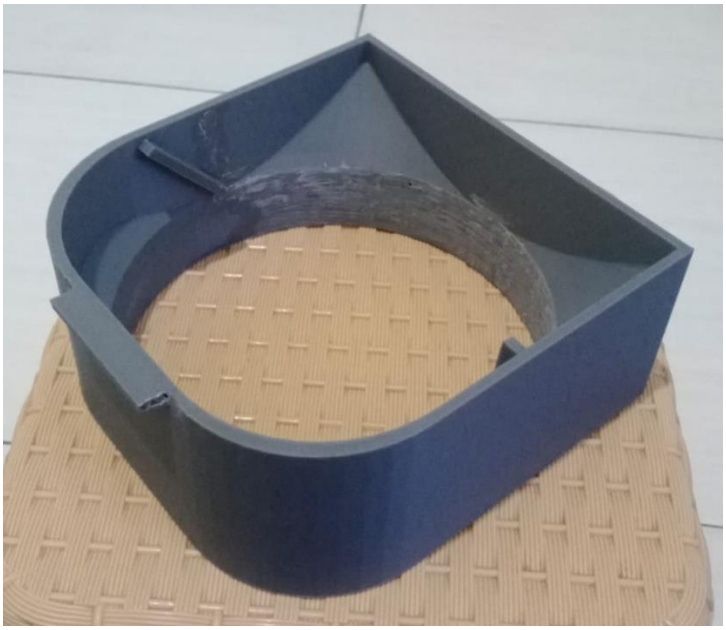
Gambar 4.7 Center Plate

4. *Rotating part* bawah yang ditunjukkan oleh Gambar 4.8, yaitu bagian yang dirancang khusus untuk berputar dan mendorong earplug keluar.



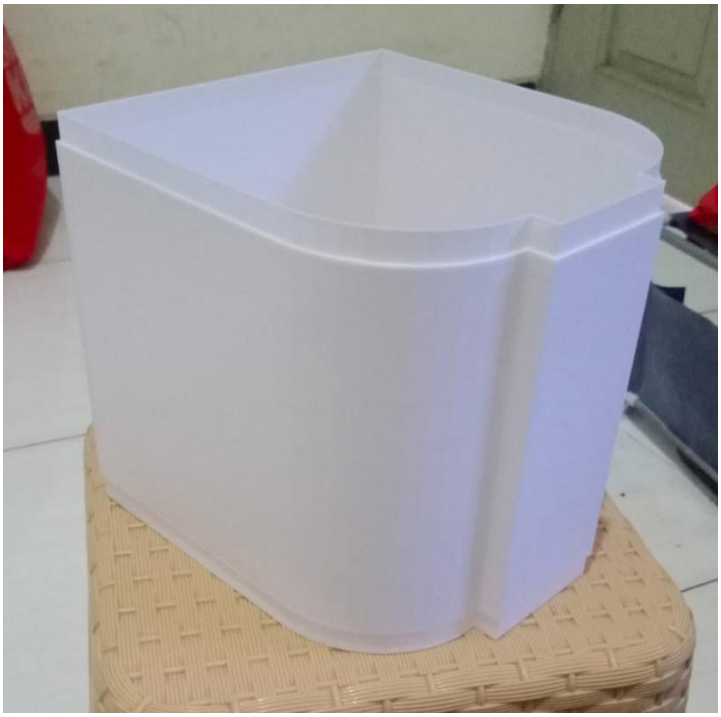
Gambar 4.8 Rotating Part

5. *Inner plate* bawah yang ditunjukkan oleh Gambar 4.9, digunakan sebagai pengarah dan penahan *earplug* agar tidak bergeser saat rotasi berlangsung.



Gambar 4.9 Inner Plate

6. *Cover* bawah yang ditunjukkan oleh Gambar 4.10, untuk menutup bagian dalam dan melindungi komponen dari debu serta kontak langsung.

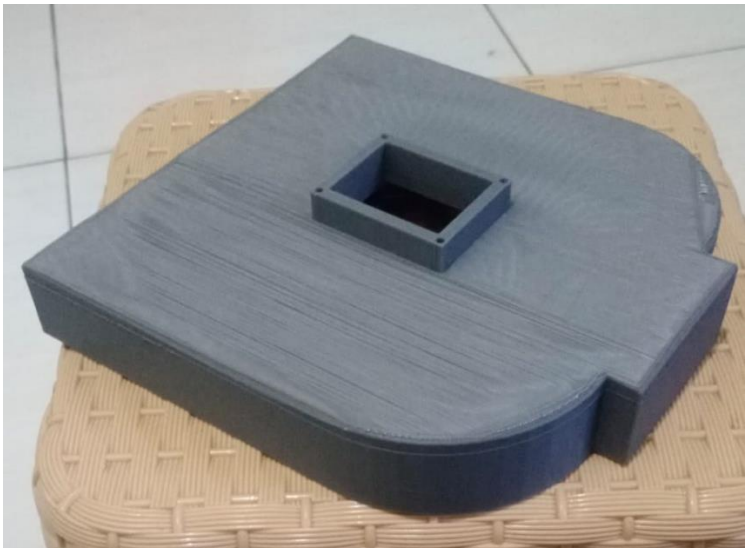


Gambar 4.10 Cover

7. *Support* tutup bawah bawah yang ditunjukkan oleh Gambar 4.11, sebagai dudukan dan pengunci bagian penutup bawah.

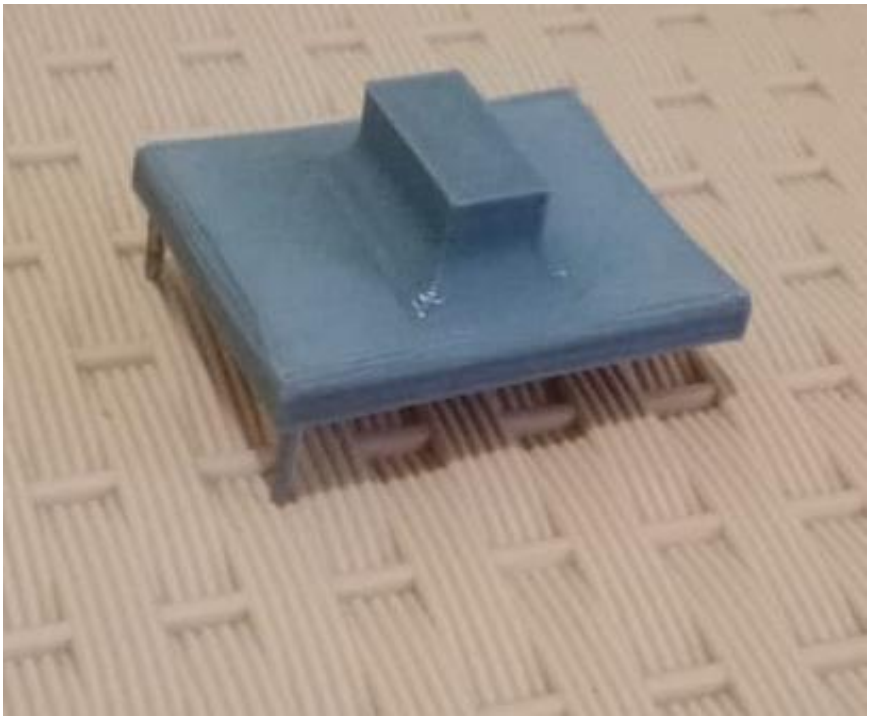
Gambar 4.11 Support Tutup Bawah

8. *Support* tutup atas bawah yang ditunjukkan oleh Gambar 4.12, sebagai penyangga dan pengunci bagian atas alat.



Gambar 4.12 Support Tutup Atas

9. Tutup atas bawah yang ditunjukkan oleh Gambar 4.13, menutup keseluruhan alat dari atas sekaligus memberikan akses untuk isi ulang *earplug*.

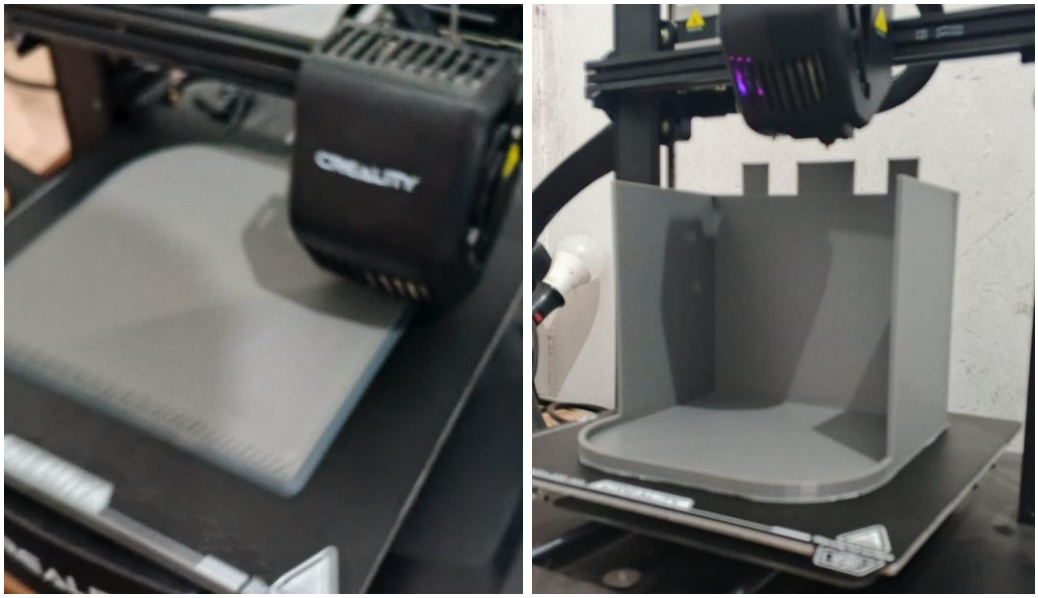


Gambar 4.13 Tutup Atas

Setiap komponen telah didesain agar dapat dirakit secara presisi menggunakan sistem *snap-fit* maupun baut, tergantung pada kebutuhan kekuatan dan fleksibilitasnya. Desain modular ini tidak hanya memudahkan proses perakitan dan perawatan, tetapi juga memungkinkan alat untuk diperluas atau dimodifikasi di masa depan.

2. Proses Pencetakan 3D Print

Desain 3D yang telah selesai kemudian dicetak menggunakan *printer 3D* dengan material PLA+. Proses pencetakan dilakukan dengan pengaturan resolusi cetak yang sesuai agar hasil cetakan memiliki presisi yang cukup tinggi, terutama pada bagian *snap-fit* dan dudukan komponen. Seluruh bagian dicetak terpisah sesuai dengan fungsi dan dirakit setelahnya. Gambar 4.14 menunjukkan proses pencetakan 3D Print.



Gambar 4.14 Proses Pembuatan 3D Print

3. Perakitan Komponen Mekanik

Setelah proses pencetakan selesai, bagian-bagian mekanik dirakit secara modular. Beberapa komponen seperti OLED, LM2596, dan servo motor dipasang menggunakan baut untuk memastikan kestabilan saat digunakan. Sementara itu, bagian casing lainnya menggunakan sistem kuncian (*snap-fit*) yang memungkinkan unit dapat dibuka dan ditutup kembali dengan mudah. Hal ini sangat bermanfaat untuk keperluan perawatan, penggantian komponen, maupun proses isi ulang earplug. Desain modular dan metode perakitan ini memungkinkan sistem dispenser memiliki fleksibilitas tinggi tanpa mengorbankan kekuatan struktur. Mekanisme pengeluaran berbasis rotasi juga memberikan kontrol yang lebih akurat terhadap jumlah *earplug* yang dikeluarkan. Gambar 4.15 menunjukkan komponen mekanik yang telah terpasang.

Gambar 4.15 Komponen Mekanik yang Telah Terpasang

4.2 Pembuatan Perangkat Lunak

Pembuatan perangkat lunak dalam proyek ini bertujuan untuk mengintegrasikan seluruh komponen sistem, baik pada sisi mikrokontroler (ESP32), pengelolaan data (*database*), hingga antarmuka pengguna melalui website dan aplikasi *mobile*. Perangkat lunak dirancang untuk memastikan bahwa proses otentikasi pengguna, pencatatan aktivitas, serta monitoring penggunaan earplug dapat dilakukan secara otomatis dan *real time*. Secara umum, perangkat lunak yang dikembangkan terbagi menjadi empat bagian utama, yaitu:

1. Program mikrokontroler menggunakan Arduino IDE dengan bahasa pemrograman C++
2. Pembuatan *database* menggunakan phpMyAdmin yang berjalan di server localhost

3. Pembuatan website *monitoring* dan kontrol menggunakan kombinasi bahasa HTML, CSS, JavaScript, dan PHP
4. Pembuatan aplikasi *mobile* menggunakan Flutter dengan bahasa Dart.

4.2.1 Pembuatan Program Pada Arduino IDE

Pemrograman mikrokontroler ESP32 dilakukan menggunakan perangkat lunak Arduino IDE dengan bahasa pemrograman C++. Program ini merupakan bagian sentral dari sistem karena mengatur kerja seluruh komponen perangkat keras seperti modul RFID, OLED display, servo motor, serta sensor TCRT5000. Selain itu, ESP32 juga menangani proses konektivitas dengan jaringan WiFi dan pertukaran data dengan *database* melalui protokol HTTP. Struktur kode dibagi ke dalam beberapa bagian utama untuk memastikan pengembangan yang modular dan efisien. Setiap bagian difokuskan pada satu fungsi spesifik agar lebih mudah dalam proses *debugging* dan pengujian. Adapun pembagian program yang dikembangkan meliputi:

1. Program untuk menghubungkan ESP32 dengan jaringan WiFi dan *database*.
2. Program untuk membaca dan memverifikasi modul RFID.
3. Program untuk menampilkan informasi pada layar OLED.
4. Program untuk mengontrol servo motor serta membaca data dari sensor TCRT5000.

4.2.1.1 Pembuatan Program Koneksi ESP32 dengan WiFi dan Database

Pada tahap ini dilakukan proses pengembangan program untuk menghubungkan ESP32 dengan jaringan WiFi dan melakukan komunikasi dengan *database server* menggunakan protokol HTTP POST dan GET. Koneksi ini penting sebagai penghubung antara sistem perangkat keras dan *server* yang menyimpan data pengguna RFID serta informasi ketersediaan *earplug*. Pada tahap awal dilakukan inisialisasi *library* dengan kode seperti pada Gambar 4.16.

```
#include <WiFi.h>
#include <HTTPClient.h>
```

Gambar 4.16 Inisialisasi Library untuk WiFi dan HTTP

Kode WiFi.h merupakan *library* ESP32 untuk menangani koneksi WiFi. Sedangkan Kode HTTPClient.h digunakan untuk membuat permintaan HTTP (GET/POST) ke server yang akan menjadi jembatan antara ESP32 dan *database* MySQL melalui file PHP. Gambar 4.17 menunjukkan kode yang digunakan sebagai variabel SSID dan Password WiFi.

```
const char* ssid = "Unknown";
const char* password = "lolah666";
```

Gambar 4.17 Kode Variabel SSID dan Password WiFi

Kode tersebut dapat disesuaikan dengan jaringan WiFi yang digunakan oleh ESP32 agar dapat terhubung ke internet. Kemudian selanjutnya diperlukan kode untuk melakukan inisialisasi koneksi WiFi seperti yang ditunjukkan pada Gambar 4.18.

```
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
  delay(2000);
  Serial.println("Connecting to WiFi...");
}
```

Gambar 4.18 Kode Untuk Melakukan Inisialisasi Koneksi WiFi Pada ESP32

ESP32 akan memulai koneksi ke jaringan WiFi menggunakan kredensial yang diberikan pada Gambar 4.18. Program akan terus menunggu hingga berhasil terkoneksi (*WL_Connected*). Pada sisi pengambilan data dari *database* ESP32 akan mengakses dua file PHP yang berada di server lokal menggunakan kode pada Gambar 4.19.

```
const char* serverName = "http://192.168.153.30/TA/Website%20(1)/check_rfid.php";
const char* earplugUrl = "http://192.168.153.30/TA/Website%20(1)/total_earplug.php";
```

Gambar 4.19 Kode Untuk Mengakses FIle PHP pada Server Lokal

Alamat 192.168.153.30 didapat dari alamat yang sama dengan koneksi WiFi pada laptop. Hal tersebut perlu diperhatikan karena jika alamat yang dituliskan berbeda, maka ESP32 tidak akan terkoneksi dengan *database*. Kemudian terdapat fungsi

sendRFIDtoServer() yang digunakan untuk mengirimkan data RFID ke *server* melalui metode POST menggunakan protokol HTTP. Kode untuk mengirim data RFID ke *server* dapat dilihat pada Gambar 4.20.

```
// Fungsi untuk mengirim UID ke server dan mendapatkan respons
String sendRFIDtoServer(String uid) {
  if (WiFi.status() == WL_CONNECTED) { // Cek apakah terhubung ke WiFi
    HTTPClient http;
    http.begin(serverName); // Tentukan URL server
    http.addHeader("Content-Type", "application/x-www-form-urlencoded"); // Set header untuk POST

    // Data yang dikirim
    String postData = "rfid_tag=" + uid;

    // Kirim permintaan POST
    int httpResponseCode = http.POST(postData);
```

Gambar 4.20 Kode Untuk Mengirim Data RFID ke Server

Selain fungsi sendRFIDtoServer() terdapat juga fungsi untuk mengambil jumlah *earplug* yang tersedia dari server. Pada proses ini digunakan fungsi getAvailableEarplug() seperti pada Gambar 4.21.

```
int getAvailableEarplug() {
  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;
    http.begin(earplugUrl); // URL untuk mengambil jumlah earplug
    int httpResponseCode = http.GET();

    if (httpResponseCode == 200) {
      String payload = http.getString();
      Serial.println("Earplug response: " + payload);

      // Parse manual karena tanpa library JSON
      int index = payload.indexOf("availableEarplug");
      if (index != -1) {
        int colonIndex = payload.indexOf(':', index);
        int endIndex = payload.indexOf('}', colonIndex);
        String valueStr = payload.substring(colonIndex + 1, endIndex);
        valueStr.trim();
        return valueStr.toInt();
      }
```

Gambar 4.21 Kode Fungsi getAvailableEarplug() pada ESP32

Pada Gambar 4.21 juga ditunjukkan proses *parsing string* yang bertujuan untuk mengambil nilai dari elemen *availableEarplug* yang terdapat dalam sebuah data berformat JSON. Proses ini diawali dengan mencari indeks keberadaan string "*availableEarplug*" pada variabel *payload* menggunakan fungsi indexOf. Jika elemen tersebut ditemukan, maka proses dilanjutkan dengan pencarian tanda titik dua (:) yang berfungsi sebagai pemisah antara kunci dan nilai dalam format JSON. Setelah titik dua ditemukan, selanjutnya dilakukan pencarian karakter penutup objek JSON, yaitu tanda kurung kurawal penutup (}), guna menentukan batas akhir dari nilai yang ingin diambil. Kemudian, digunakan fungsi *substring* untuk mengekstrak nilai yang berada di antara tanda titik dua dan tanda kurung kurawal penutup tersebut. Hasil ekstraksi berupa string disimpan dalam variabel *valueStr*, yang kemudian dibersihkan dari kemungkinan spasi menggunakan metode *trim*. Terakhir, nilai *string* yang telah bersih dikonversi menjadi bilangan bulat melalui fungsi toInt dan dikembalikan sebagai hasil akhir dari proses *parsing* tersebut. Teknik ini merupakan bentuk *parsing* manual yang umum digunakan dalam pemrograman Arduino ketika tidak menggunakan pustaka khusus pengelola JSON seperti ArduinoJson.

4.2.1.2 Pembuatan Program Modul RFID

Pada tahap ini dilakukan pembuatan program untuk modul RFID berbasis ESP32 yang berfungsi sebagai media identifikasi pengguna melalui kartu *tag* RFID. Modul RFID yang digunakan adalah Modul MFRC522 yang bekerja pada frekuensi 13,56 MHz dan berkomunikasi dengan mikrokontroler melalui SPI (*Serial Peripheral Interface*). Langkah awal dalam pemrograman adalah mengimpor atau menginisalisasi *library* yang diperlukan. Pada Gambar 4.22 ditunjukkan *library* yang digunakan untuk komunikasi antara ESP32 dan MFRC522.

```
#include <MFRC522.h>
```

Gambar 4.22 Inisialisasi Library untuk Modul RFID

Selanjutnya dilakukan deklarasi pin yang digunakan untuk menghubungkan ESP32 dengan modul RFID seperti pada Gambar 4.23.

```
// RFID Pin setup
#define SS_PIN 4  // SDA ke D4
#define RST_PIN 5 // RST ke D5
#define SCK 18    // SCK ke D18
#define MOSI 23   // MOSI ke D23
#define MISO 19   // MISO ke D19
MFRC522 rfid(SS_PIN, RST_PIN);  // Instance RFID
```

Gambar 4.23 Kode yang Menunjukkan Deklarasi Pin untuk Modul RFID

Objek RFID dibuat dengan *syntax* MFRC522 rfid(SS_PIN, RST_PIN). Protokol komunikasi diinisialisasi secara manual seperti pada Gambar 4.24.

```
// Inisialisasi SPI secara manual untuk RFID
SPI.begin(SCK, MISO, MOSI, SS_PIN); // Pin SPI manual
rfid.PCD_Init(); // Inisialisasi RFID
Serial.println("RFID ready");
```

Gambar 4.24 Protokol Komunikasi RFID dan ESP32

Hal tersebut dilakukan untuk memastikan bahwa koneksi antar perangkat telah tersinkronisasi dengan benar karena modul RFID MFRC522 tidak terhubung ke pin SPI *default* pada ESP32. Kemudian fungsi *rfid.PCD_Init()* dipanggil untuk mengaktifkan dan menginisialisasi modul MFRC522 sehingga siap mendeteksi keberadaan *tag* RFID.

Dalam fungsi *loop()* seperti pada Gambar 4.25, sistem secara terus menerus memantau keberadaan karti RFID menggunakan perintah *rfid.PICC_IsNewCardPresent()*.

```
// Cek keberadaan kartu RFID
Serial.println("Checking for RFID tag...");
if (rfid.PICC_IsNewCardPresent()) {
  Serial.println("Card detected!");
  if (rfid.PICC_ReadCardSerial()) {
    Serial.println("Reading card...");
    String rfidUID = "";
    for (byte i = 0; i < rfid.uid.size; i++) {
      rfidUID += String(rfid.uid.uidByte[i], HEX);
    }
    Serial.println("UID tag RFID: " + rfidUID);
```

Gambar 4.25 Kode Pemindaian dan Pembacaan Kartu Tag RFID

Apabila terdapat kartu yang terdeteksi, maka dilanjutkan dengan proses pembacaan UID (*Unique Identifier*) dari kartu tersebut menggunakan *rfid.PICC_ReadCardSerial()*. UID tersebut merupakan kode unik dari setiap kartu dan dikonversi menjadi string heksadesimal yang dapat dikirim ke *server* melalui jaringan WiFi. UID ini dikonstruksi dalam bentuk string dengan menggunakan perulangan *for* berdasarkan *rfid.uid.size*.

```
// Fungsi untuk mengirim UID ke server dan mendapatkan respons
String sendRFIDtoServer(String uid) {
  if (WiFi.status() == WL_CONNECTED) { // Cek apakah terhubung ke WiFi
    HTTPClient http;
    http.begin(serverName); // Tentukan URL server
    http.addHeader("Content-Type", "application/x-www-form-urlencoded"); // Set header untuk POST

    // Data yang dikirim
    String postData = "rfid_tag=" + uid;

    // Kirim permintaan POST
    int httpResponseCode = http.POST(postData);

    String response = "";
    if (httpResponseCode > 0) {
      response = http.getString(); // Dapatkan respons dari server
      Serial.println("Server response: " + response);
    } else {
      Serial.println("Error on sending POST: " + String(httpResponseCode));
    }

    http.end(); // Tutup koneksi
    return response;
  } else {
    Serial.println("Error in WiFi connection");
    return "fail";
  }
}
```

Gambar 4.26 Fungsi sendRFIDtoServer()

Setelah UID berhasil dibaca, maka fungsi *sendRFIDtoServer()* akan dipanggil seperti pada Gambar 4.26. Fungsi ini akan membuat koneksi HTTP POST ke URL *server* yang telah ditentukan dalam variabel *serverName*. Data yang dikirim berupa pasangan *key-value* dengan format *rfid_tag=<UID>*. Selanjutnya, respons dari server akan dianalisis. Terdapat tiga kemungkinan tanggapan dari *server*, yaitu "*success*", "*limit_exceeded*", dan "*invalid_card*".

4.2.1.3 Pembuatan Program OLED

Pada tahap ini dilakukan implementasi tampilan visual menggunakan layar OLED sebagai media interaksi antarmuka pengguna pada sistem. Untuk mendukung penggunaan perangkat keras OLED, terlebih dahulu diimpor dua buah *library* utama, yaitu *Adafruit_GFX.h* dan *Adafruit_SSD1306.h*, yang masing-masing berfungsi untuk pengelolaan grafis dasar serta pengendalian langsung terhadap modul layar OLED berbasis chipset SSD1306. Selain itu, pustaka *Wire.h*

turut dimuat untuk menunjang komunikasi I2C yang merupakan metode komunikasi utama antara mikrokontroler ESP32 dengan modul OLED. Gambar 4.27 menunjukkan program dalam melakukan inisialisasi *library* OLED.

```
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
```

Gambar 4.27 Inisialisasi Library OLED

Selanjutnya, ditentukan parameter dasar OLED yaitu lebar (128 piksel) dan tinggi (64 piksel) layar melalui *#define SCREEN_WIDTH* dan *SCREEN_HEIGHT*, serta konfigurasi pin I2C khusus ESP32 dengan mendefinisikan OLED_SDA pada pin 21 dan OLED_SCL pada pin 22. Nilai OLED_RESET diatur ke -1 karena pada modul OLED yang digunakan tidak tersedia pin reset terpisah. Seluruh parameter tersebut dikompilasikan ke dalam objek *display* bertipe *Adafruit_SSD1306* yang akan digunakan untuk seluruh pengendalian tampilan. Proses penentuan parameter dan kompilasi objek *display* ditunjukkan pada Gambar 4.28.

```
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
#define OLED_SDA 21
#define OLED_SCL 22
#define OLED_RESET -1
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);
```

Gambar 4.28 Proses Penentuan Parameter dan Kompilasi Objek Display

Pada fungsi *setup(),* dilakukan inisialisasi komunikasi I2C melalui perintah *Wire.begin(OLED_SDA, OLED_SCL)*, kemudian dilanjutkan dengan pemanggilan fungsi *display.begin()* untuk memulai penggunaan OLED. Dalam proses inisialisasi ini, juga disisipkan kondisi pengecekan keberhasilan alokasi memori layar; apabila gagal, maka sistem akan berhenti berjalan dengan perulangan tak berhingga. Setelah inisialisasi berhasil, layar dibersihkan menggunakan *display.clearDisplay(),* dan dilakukan penulisan teks *"Connecting to WiFi..."* sebagai umpan balik visual bahwa sistem tengah mencoba menghubungkan diri ke

jaringan nirkabel. Proses inisalisasi I2C untuk OLED hingga penulisan teks "*Connecting WiFi*" ditunjukkan pada Gambar 4.29.

```
// Inisialisasi I2C untuk OLED
Wire.begin(OLED_SDA, OLED_SCL);

// Inisialisasi OLED
if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
  Serial.println(F("OLED allocation failed"));
  for (;;); // Berhenti di sini jika gagal
}

// Bersihkan display dan tampilkan status WiFi
display.clearDisplay();
display.setTextSize(1);
display.setTextColor(SSD1306_WHITE);
display.setCursor(0, 30);
display.println("Connecting to WiFi...");
display.display();
```

Gambar 4.29 Proses Inisialisasi I2C untuk OLED dan Penulisan Text

Apabila koneksi WiFi berhasil dilakukan, tampilan OLED diperbarui untuk menampilkan pesan "*WiFi Connected*!", dengan jeda waktu 2 detik sebagai jeda visual sebelum memasuki logika utama. Program penampilan pesan WiFi ditunjukkan oleh Gambar 4.30. Selama eksekusi fungsi *loop()*, layar OLED terus diperbarui secara dinamis. Layar secara rutin menampilkan status permintaan pengguna untuk men-*scan* kartu RFID dengan menampilkan pesan "*Please Scan*" dan "*Your Card*". Program menampilkan pesan tersebut ditunjukkan oleh Gambar 4.31. Selain itu, layar juga digunakan untuk menampilkan informasi jumlah earplug yang tersedia. Nilai tersebut didapat melalui fungsi *getAvailableEarplug()* yang melakukan permintaan HTTP GET ke *server*, dan hasilnya ditampilkan di bagian bawah layar OLED. Program menampilkan jumlah *earplug* yang tersedia ditampilkan oleh Gambar 4.32.

```
// Jika berhasil terhubung ke WiFi
Serial.println("WiFi Connected");
display.clearDisplay();
display.setCursor(20, 30);
display.println("WiFi Connected!");
display.display();
delay(2000);
```

Gambar 4.30 Kode Penampilan Pesan Status WiFi

```
// Tampilkan pesan scan
display.setCursor(30, 15);
display.println("Please Scan");
display.setCursor(35, 35);
display.println("Your Card");
display.display();
```

Gambar 4.31 Kode Penampilan Pesan Permintaan Scan Tag RFID

```
// Tampilkan jumlah earplug
int earplugCount = getAvailableEarplug();
display.setCursor(30, 55);
if (earplugCount >= 0) {
  display.print("Earplug: ");
  display.println(earplugCount);
} else {
  display.println("Server Error"); // Jika gagal ambil data
}
```

Gambar 4.32 Program Penampilan Jumlah Earplug Tersedia Pada OLED

Pada saat *tag* RFID berhasil terbaca dan *server* memberikan respons berupa *success*, tampilan pada layar OLED akan menampilkan pesan "Silakan Ambil" sebagai instruksi kepada pengguna bahwa proses pengambilan earplug dapat

dilakukan seperti pada Gambar 4.33. Sebaliknya, apabila server merespons dengan *limit_exceeded*, yang menandakan bahwa pengguna telah mencapai batas maksimum pengambilan, maka layar akan menampilkan pesan "Akses Ditolak" disertai keterangan "Limit Habis" seperti pada Gambar 4.34. Sementara itu, jika server memberikan respons *invalid_card*, yang mengindikasikan bahwa data pengguna tidak ditemukan dalam basis data, maka OLED akan menampilkan pesan "Akses Ditolak" beserta keterangan "Data Tidak Ditemukan" seperti pada Gambar 4.35.

```
String serverResponse = sendRFIDtoServer(rfidUID);
if (serverResponse == "success") {
  // Jika terdaftar, tampilkan "Silakan Ambil" dan putar servo searah jarum jam
  display.clearDisplay();
  display.setTextSize(1);
  display.setCursor(25, 30);
  display.println("Silakan Ambil");
  display.display();
  rotateServoClockwise();
```

Gambar 4.33 Kode Pada ESP32 Apabila Respon Server "success"

```
  stopServo(); // Hentikan pergerakan servo setelah terdeteksi 2 benda
} else if (serverResponse == "limit_exceeded") {
  // Jika limit terpenuhi, tampilkan akses ditolak dan pesan limit
  display.clearDisplay();
  display.setTextSize(1);
  display.setCursor(30, 20);
  display.println("Akses Ditolak");
  display.setCursor(35, 40);
  display.println("Limit Habis");
  display.display();
```

Gambar 4.34 Kode Pada ESP32 Apabila Respon Server "limit_exceedeed"

```
} else if (serverResponse == "invalid_card") {
  // Jika kartu tidak terdaftar, tampilkan akses ditolak dan data tidak ditemukan
  display.clearDisplay();
  display.setTextSize(1);
  display.setCursor(30, 20);
  display.println("Akses Ditolak");
  display.setCursor(5, 40);
  display.println("Data Tidak Ditemukan");
  display.display();
```

Gambar 4.35 Kode Pada ESP32 Apabila Respon Server "invalid_card"

4.2.1.4 Pembuatan Program Kontrol Servo dan TCRT5000

Pada tahap ini, dijelaskan proses pembuatan program mikrokontroler untuk mengatur servo motor dan membaca sensor TCRT5000 sebagai sistem aktuator dan deteksi benda pada alat peminjaman *earplug* otomatis. *Servo* digunakan sebagai penggerak mekanis untuk membuka akses pengambilan *earplug*. Servo dikendalikan melalui pin digital (pin 13) dengan bantuan *library* ESP32Servo. Gambar 4.36 menunjukkan inisialisasi *library* untuk servo sedangkan Gambar 4.37 menunjukkan inisialisasi pin yang digunakan oleh servo.

```
#include <ESP32Servo.h>
```

Gambar 4.36 Library Servo Pada Arduino IDE

```
// Servo setup
#define SERVO_PIN 13 // Pin servo
Servo myServo;       // Membuat objek servo
```

Gambar 4.37 Inisialisasi Pin Servo

Servo diinisialisasi pada posisi netral (*myServo.write(90)*) agar dalam kondisi *standby* saat sistem menyala. Karena jenis servo yang digunakan adalah servo *continous* maka ketika servo diberi program *myServo.write(90)* servo akan berhenti. Program untuk membuat servo standby ditunjukkan oleh Gambar 4.38.

```
// Inisialisasi Servo
myServo.attach(SERVO_PIN); // Menghubungkan servo ke pin
myServo.write(90); // Servo mulai pada posisi netral
```

Gambar 4.38 Kode Untuk Membuat Servo Standby

Saat kartu RFID pengguna dikenali, mikrokontroler ESP32 akan mengirimkan data UID ke *server* melalui koneksi Wi-Fi dan menunggu respon *server* dalam bentuk data JSON. *Server* akan mengevaluasi UID tersebut dan merespons dengan status akses. Jika respon *server* menyatakan akses "*success*", maka sistem lokal akan menjalankan prosedur pengeluaran *earplug*. Respon ini menjadi kunci pemicu aktifnya servo motor untuk membuka saluran pengeluaran. Perintah *rotateServoClockwise()* kemudian dijalankan untuk menggerakkan servo dari posisi diam (90°) menjadi berputar searah dengan jarum jam (80°), yang secara mekanik membuka jalur pengeluaran earplug. Servo *continous* hanya dapat bergerak searah jarum jam atau berlawanan arah jarum jam. Untuk mengontrol arah tersebut digunakan nilai *myServo.write()*, jika nilai kurang dari 90° maka servo akan berputar searah jarum jam dan jika nilai lebih dari 90° servo akan berputar berlawanan arah jarum jam. Setelah gerakan ini, sensor TCRT5000 akan aktif mendeteksi setiap *earplug* yang melewati jalurnya. Sensor ini bekerja dengan membaca pantulan cahaya infra merah, di mana output LOW dan nilai analog <800 menunjukkan keberadaan objek di depannya. Sistem akan menghitung dua kali deteksi sukses oleh sensor untuk memastikan bahwa dua buah earplug telah dikeluarkan.

```
// Kirim UID ke server dan tampilkan respons
String serverResponse = sendRFIDtoServer(rfidUID);
if (serverResponse == "success") {
  // Jika terdaftar, tampilkan "Silakan Ambil" dan putar servo searah jarum jam
  display.clearDisplay();
  display.setTextSize(1);
  display.setCursor(25, 30);
  display.println("Silakan Ambil");
  display.display();
  rotateServoClockwise();

  int objectCount = 0;
  while (objectCount < 2) {
    int digitalValue = digitalRead(TCRT_D0_PIN); // Pembacaan D0 secara digital
    int analogValue = analogRead(TCRT_A0_PIN); // Pembacaan A0 secara analog

    if (digitalValue == LOW && analogValue < 800) {
      objectCount++;
      Serial.println("Object detected! Count: " + String(objectCount));
      delay(500);  // Delay untuk menghindari double count
      while (digitalRead(TCRT_D0_PIN) == LOW || analogRead(TCRT_A0_PIN) < 800) {
      }
    } else {
      Serial.println("No valid object detected.");
    }
  }
```

Gambar 4.39 Kode Logika untuk Modul TCRT5000 Dan Servo

Pada Gambar 4.39 ditunjukkan logika yang diberikan kepada modul TCRT5000 dan servo. Pada sisi modul TCRT5000 program ini dimulai dengan inisialisasi variabel *objectCount* yang berfungsi sebagai penghitung jumlah objek yang terdeteksi. Selama nilai *objectCount* masih kurang dari dua, program akan terus melakukan pembacaan sensor secara berulang. Sensor TCRT5000 memiliki dua jenis keluaran, yaitu digital dan analog. Pembacaan digital dilakukan melalui pin TCRT_D0_PIN menggunakan fungsi *digitalRead*, sedangkan pembacaan analog dilakukan melalui pin TCRT_A0_PIN menggunakan fungsi *analogRead*. Deteksi objek dianggap valid apabila nilai digital menunjukkan kondisi *LOW* dan nilai analog lebih kecil dari 800, yang menandakan bahwa terdapat pantulan cahaya dari objek yang cukup kuat dan dekat. Jika kondisi tersebut terpenuhi, maka penghitung objek akan ditambahkan satu dan pesan keberhasilan deteksi objek akan dicetak pada *serial monitor*. Selanjutnya, program menambahkan jeda selama 500 milidetik menggunakan fungsi *delay* untuk mencegah terjadinya penghitungan

ganda terhadap objek yang sama akibat pergerakan yang lambat atau pembacaan sensor yang terlalu sensitif.

Setelah itu, program masuk ke dalam *loop* penahan, yaitu *while*, yang bertugas memastikan bahwa sistem menunggu hingga objek benar-benar telah keluar dari jangkauan sensor sebelum melakukan deteksi berikutnya. Hal ini dilakukan dengan terus memeriksa apakah keluaran digital masih *LOW* atau keluaran analog masih di bawah 800. Apabila kedua kondisi tersebut tidak lagi terpenuhi, maka sistem keluar dari *loop* penahan dan siap mendeteksi objek selanjutnya. Sebaliknya, jika pada awal pembacaan tidak ditemukan objek valid, maka program akan mencetak pesan "*No valid object detected*" di *serial monitor* sebagai bentuk umpan balik kepada pengguna. Struktur kode ini secara keseluruhan dirancang untuk menghasilkan sistem deteksi objek yang akurat dan menghindari kesalahan penghitungan akibat pembacaan ganda dari objek yang sama.

4.2.2 Pembuatan Database

Dalam proses pengembangan sistem, pembuatan *database* menjadi salah satu tahap krusial yang berfungsi untuk menyimpan, mengelola, serta mengorganisir data yang dibutuhkan dalam sistem. Pada sistem ini, terdapat dua *database* utama yang dibangun untuk mendukung proses autentikasi pengguna dan penyimpanan data operasional. *Database* pertama diberi nama *user_management*, yang dirancang khusus untuk mengelola proses *login* dan *register* bagi pengguna yang memiliki otorisasi sebagai *admin*. Tabel utama dalam *database* ini adalah users, yang menyimpan informasi seperti nama lengkap, nomor induk, nama pengguna (*username*), dan kata sandi (*password*).

Sementara itu, *database* kedua dinamakan earplug_system, yang berfungsi sebagai wadah penyimpanan data historis dan operasional sistem, seperti data pemakaian dan pengisian ulang (*refill*) *earplug*, serta data pengguna yang menggunakan teknologi RFID. *Database* ini terdiri dari beberapa tabel, antara lain *rfid_users, history*, dan *earplug_refills*. Tabel-tabel tersebut saling terhubung melalui relasi kunci asing (*foreign key*), sehingga memastikan integritas dan keterhubungan data secara konsisten. Dengan perancangan struktur basis data yang

sistematis dan sesuai dengan kebutuhan sistem, pengelolaan data menjadi lebih efisien dan mampu mendukung kinerja sistem secara keseluruhan, baik untuk versi *web* maupun *mobile application*.

4.2.2.1 Pembuatan Database untuk Login dan Register Page

Pembuatan *database login* dan *register* bertujuan untuk mengelola proses autentikasi pengguna pada sistem, baik melalui platform *website* maupun aplikasi *mobile*. Untuk menyimpan data autentikasi, digunakan *database* dengan nama *user_management*, yang memiliki satu tabel utama bernama *users*. Kolom tabel *users* pada *database user_management* ditunjukkan oleh Gambar 4.40.

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	int(11)			No	None		AUTO_INCREMENT	Change Drop More
2	firstname	varchar(100)	utf8mb4_general_ci		No	None			Change Drop More
3	lastname	varchar(100)	utf8mb4_general_ci		No	None			Change Drop More
4	no_induk	varchar(100)	utf8mb4_general_ci		No	None			Change Drop More
5	username	varchar(100)	utf8mb4_general_ci		No	None			Change Drop More
6	password	varchar(255)	utf8mb4_general_ci		No	None			Change Drop More

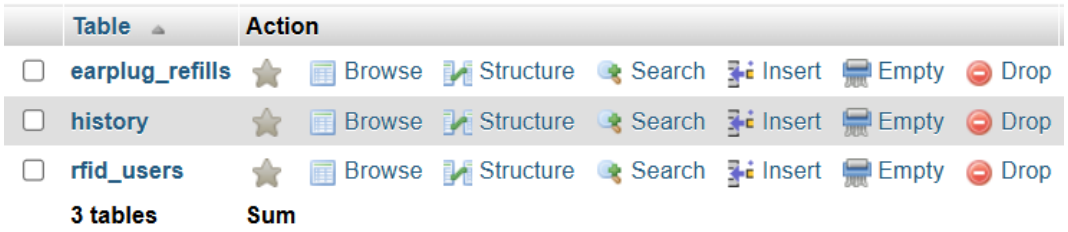
Gambar 4.40 Kolom Pada Tabel Users

Tabel *users* terdiri atas beberapa kolom yang memiliki fungsi spesifik untuk mendukung proses registrasi dan *login admin*. Kolom *firstname* digunakan untuk menyimpan nama awal dari *admin*, sedangkan *lastname* menyimpan nama akhir dari *admin*. Kolom no_induk berfungsi sebagai penyimpan nomor induk setiap *admin* yang terdaftar. Selanjutnya, kolom *username* digunakan untuk membuat nama pengguna yang akan digunakan saat proses *login*, dan kolom *password* berfungsi untuk menyimpan kata sandi yang digunakan *admin* untuk masuk ke dalam sistem.

Untuk menjaga keunikan data, kolom no_induk dan username diberikan batasan *UNIQUE*, sedangkan kolom id digunakan sebagai *primary key* dan bersifat *auto increment* guna memastikan setiap entri memiliki identitas yang unik dan tidak duplikatif.

4.2.2.2 Pembuatan Database untuk Dashboard

Pembuatan *database* ini dilakukan untuk menyimpan dan mengelola data yang berkaitan dengan pengguna RFID, riwayat penggunaan *earplug*, serta data pengisian ulang *earplug* oleh *admin*. Pada tahap ini digunakan *database* dengan nama *earplug_system*, yang terdiri atas tiga tabel utama, yaitu *rfid_users*, *history*, dan *earplug_refills*. Tabel pada *database earplug_system* ditunjukkan oleh Gambar 4.41.



Gambar 4.41 Tabel Pada Database earplug_system

Tabel *rfid_users* berfungsi untuk menyimpan data pengguna yang menggunakan kartu RFID dalam pengambilan *earplug*. Kolom *rfid_id* berperan sebagai identitas unik tiap entri dalam tabel. Kolom *no_induk* digunakan untuk menyimpan nomor identifikasi pengguna, sedangkan *rfid_tag* menyimpan data tag RFID yang terhubung dengan pengguna tersebut. Kolom nama berisi nama lengkap pengguna dan divisi berisi informasi mengenai divisi atau departemen tempat pengguna bekerja. Selain itu, *tap_limit* menentukan jumlah maksimal pengambilan *earplug* oleh pengguna dalam satu hari. Kolom *active_status* menunjukkan status aktif atau tidaknya pengguna dalam sistem, dan *created_at* mencatat waktu pertama kali data pengguna ditambahkan ke dalam sistem. Struktur tabel *rfid_users* ditampilkan pada Gambar 4.42.

	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
☐	1	**rfid_id**	int(11)			No	*None*		AUTO_INCREMENT	Change Drop More
☐	2	**no_induk**	varchar(225)	utf8mb4_general_ci		No	*None*			Change Drop More
☐	3	**rfid_tag**	varchar(255)	utf8mb4_general_ci		No	*None*			Change Drop More
☐	4	**nama**	varchar(255)	utf8mb4_general_ci		No	*None*			Change Drop More
☐	5	**divisi**	varchar(255)	utf8mb4_general_ci		No	*None*			Change Drop More
☐	6	**tap_limit**	int(11)			Yes	2			Change Drop More
☐	7	**active_status**	varchar(50)	utf8mb4_general_ci		Yes	active			Change Drop More
☐	8	**created_at**	timestamp			No	current_timestamp()			Change Drop More

Gambar 4.42 Struktur Kolom Pada Tabel rfid_users

Selanjutnya, tabel *history* digunakan untuk mencatat setiap aktivitas pengambilan *earplug* oleh pengguna. Kolom id adalah identitas unik untuk setiap entri riwayat. Kolom *name*, no_induk, dan *department* masing-masing menyimpan nama pengguna, nomor induk, serta departemen asal pengguna. Kolom *date* dan *time* mencatat tanggal dan waktu pengambilan *earplug*, sedangkan *rfid_tag* mencatat *tag* RFID yang digunakan dalam transaksi tersebut. Kolom *used_quantity* menyimpan jumlah *earplug* yang diambil, dan *timestamp* mencatat waktu otomatis saat data disimpan. Kolom status berfungsi untuk menunjukkan status transaksi, misalnya normal atau tidak normal, sedangkan *updated_at* mencatat waktu terakhir entri tersebut diperbarui secara otomatis. Struktur tabel *history* ditunjukkan oleh Gambar 4.43.

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
1	**id**	int(11)			No	*None*		AUTO_INCREMENT
2	**name**	varchar(255)	utf8mb4_general_ci		Yes	*NULL*		
3	**no_induk**	varchar(225)	utf8mb4_general_ci		Yes	*NULL*		
4	**department**	varchar(255)	utf8mb4_general_ci		Yes	*NULL*		
5	**date**	date			Yes	*NULL*		
6	**time**	time			Yes	*NULL*		
7	**rfid_tag**	varchar(255)	utf8mb4_general_ci		Yes	*NULL*		
8	**used_quantity**	int(11)			Yes	*NULL*		
9	**timestamp**	timestamp			No	current_timestamp()		
10	**status**	varchar(50)	utf8mb4_general_ci		Yes	normal		
11	**updated_at**	timestamp			No	current_timestamp()		ON UPDATE CURRENT_TIMESTAMP()

Gambar 4.43 Struktur Kolom Pada Tabel History

Tabel terakhir adalah *earplug_refills*, yang digunakan untuk mencatat setiap pengisian ulang *earplug* oleh *admin*. Kolom *refill_id* berfungsi sebagai identitas

unik setiap entri pengisian. Kolom *admin_id* menyimpan identifikasi *admin* yang melakukan pengisian ulang, sedangkan *earplug_count* menunjukkan jumlah *earplug* yang ditambahkan ke dalam mesin. Kolom *refill_date* mencatat waktu pengisian secara otomatis, dan *remaining_earplugs* menyimpan jumlah *earplug* yang tersisa setelah pengisian. Struktur tabel *earplug_refills* ditampilkan pada Gambar 4.44.

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
1	**refill_id** 🔑	int(11)			No	*None*		AUTO_INCREMENT
2	**admin_id**	int(11)			Yes	*NULL*		
3	**earplug_count**	int(11)			Yes	*NULL*		
4	**refill_date**	timestamp			No	current_timestamp()		
5	**remaining_earplugs**	int(11)			Yes	*NULL*		

Gambar 4.44 Struktur Kolom Pada Tabel earplug_refills

Dengan adanya struktur *database* ini, sistem dapat menjalankan fungsi *monitoring*, pelacakan riwayat penggunaan, dan manajemen inventaris *earplug* secara efisien dan terintegrasi antara *website* dan aplikasi *mobile*.

4.2.3 Pembuatan Program Website

Pembuatan program *website* bertujuan untuk menyediakan sistem *monitoring* sekaligus pengelolaan akses terhadap pengambilan *earplug* yang dilakukan oleh pengguna melalui kartu RFID. Website ini dirancang secara khusus hanya untuk diakses oleh *admin*, dan berfungsi sebagai antarmuka untuk mengatur batas pengambilan *earplug*, memantau riwayat penggunaan dan pengisian, serta mengelola data *user* yang diperbolehkan untuk mengambil *earplug*. *Website* dikembangkan dengan memanfaatkan kombinasi HTML, JavaScript, dan CSS untuk tampilan antarmuka (*frontend*), serta PHP untuk pengelolaan logika *backend*. Koneksi ke *database* dilakukan menggunakan PDO (PHP *Data Objects*), yang memberikan fleksibilitas dan keamanan dalam pengolahan data. Subbab-subab berikut akan membahas secara terperinci tahapan pembuatan masing-masing halaman pada sistem *website* ini.

4.2.3.1 Pembuatan Program Login Page pada Website

Pada pembuatan halaman login, struktur utama antarmuka pengguna dirancang menggunakan *HyperText Markup Language* (HTML). HTML berperan sebagai kerangka dasar dari tampilan *web*, di mana elemen-elemen seperti formulir, kotak *input*, dan tombol didefinisikan. Dalam implementasi ini, *form login* dibungkus dalam sebuah elemen <*form*> yang memiliki atribut *action* dan *method*. Atribut *action* merujuk pada alamat *file process_login.php*, yaitu *file* PHP yang akan memproses data *login* yang dikirimkan oleh pengguna. Sementara itu, atribut *method* bernilai *POST*, yang menunjukkan bahwa data yang dikirim tidak akan ditampilkan secara eksplisit pada *URL*, sehingga lebih aman dibandingkan metode *GET*. Gambar 4.45 menunjukkan struktur *form login* pada *Login Page.html*.

```
<div class="form-box">
    <div class="login-container" id="login">
        <div class="top">
            <header>Login</header>
        </div>

        <form action="http://localhost/TA/Website%20(1)/process_login.php" method="POST">
            <div class="input-box">
                <input type="text" name="username" class="input-field" placeholder="Username or Email" required>
                <i class="bx bx-user"></i>
            </div>
            <div class="input-box">
                <input type="password" name="password" class="input-field" placeholder="Password" required>
                <i class="bx bx-lock-alt"></i>
            </div>
            <div class="input-box">
                <input type="submit" class="submit" value="Sign In">
            </div>
            <div class="top">
                <span>Don't have an account? <a href="#" onclick="register()">Sign Up</a></span>
            </div>
        </form>
    </div>
```

Gambar 4.45 Struktur Form Login Pada Login Page.html

Di dalam elemen <*form*> tersebut, terdapat dua komponen utama berupa *input field* untuk *username* dan *password*. Masing-masing *field* didefinisikan menggunakan elemen <*input*> dengan tipe *text* untuk *username* dan *password* untuk kata sandi. Masing-masing *field* juga memiliki atribut name yang berfungsi sebagai penanda untuk pengolahan data di sisi *server*. Atribut *placeholder* digunakan untuk memberikan petunjuk kepada pengguna mengenai data yang harus diisikan pada kolom tersebut. Selain itu, setiap *input* dibungkus dalam sebuah <*div*> dengan *class input-box*, yang digunakan untuk keperluan penataan tampilan

oleh CSS. Ikon yang menyertai setiap *input* menggunakan elemen <*i*> dari pustaka *ikon Boxicons*, yang menambah kejelasan fungsi tiap kolom *input*.

Selanjutnya, terdapat elemen <*input*> bertipe *submit* yang berfungsi sebagai tombol untuk mengirimkan data *login*. Tombol ini diberi *class submit* dan label berupa teks "*Sign In*", yang secara visual akan ditampilkan sebagai tombol utama pada halaman *login*.

```php
include 'db_connection.php';
session_start(); // Memulai session untuk menyimpan pesan error

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $username = htmlspecialchars($_POST['username']); // Sanitasi input username
    $password = $_POST['password']; // Ambil password dari form

    try {
        // Ambil data user dari database berdasarkan username
        $sql = "SELECT * FROM users WHERE username = :username";
        $stmt = $conn->prepare($sql);
        $stmt->bindParam(':username', $username);
        $stmt->execute();
        $user = $stmt->fetch(PDO::FETCH_ASSOC);

        // Jika user ditemukan dan password cocok
        if ($user && $password === $user['password']) {
            // Redirect ke halaman dashboard
            header("Location: http://localhost/TA/Website%20(1)/Dashboard.html");
            exit; // Menghentikan eksekusi setelah redirect
        } else {
            // Menyimpan pesan error ke dalam session untuk ditampilkan di halaman login
            $_SESSION['error'] = 'Invalid username or password.';
            header("Location: http://localhost/TA/Website%20(1)/Login%20Page.html"); // Tetap di halaman login
            exit;
        }
    } catch (PDOException $e) {
        $_SESSION['error'] = 'Error: ' . $e->getMessage();
        header("Location: http://localhost/TA/Website%20(1)/Login%20Page.html");
        exit;
    }
}
```

Gambar 4.46 Kode process_login.php

Gambar 4.46 menujukkan kode program *process_login.php* yang merupakan skrip sisi *server (server side)* yang berfungsi untuk memproses autentikasi pengguna pada saat login. Skrip ini dimulai dengan menyertakan berkas *db_connection.php* menggunakan perintah *include*, yang bertujuan untuk membuka koneksi ke *database* melalui konfigurasi yang telah ditentukan sebelumnya. Setelah itu, fungsi *session_start()* dipanggil untuk memulai sesi PHP, sehingga *server* dapat menyimpan data pengguna atau pesan kesalahan selama interaksi berlangsung.

Selanjutnya, sistem melakukan pengecekan terhadap metode permintaan HTTP. Apabila permintaan yang diterima merupakan metode *POST*, maka skrip

akan mengeksekusi blok logika utama untuk proses *login*. Data yang dikirim melalui *form* akan ditangkap dengan menggunakan *$_POST*. Untuk mencegah potensi serangan *cross-site scripting* (XSS), nilai dari kolom *username* dipecah menjadi karakter khusus terlebih dahulu menggunakan fungsi *htmlspecialchars(),* sedangkan *password* diambil secara langsung.

Proses utama autentikasi dilakukan melalui pemanggilan *query databases* yang menyeleksi seluruh data dari tabel *users* berdasarkan *username* yang dimasukkan oleh pengguna. *Query* ini disiapkan menggunakan metode *prepare()* dari objek koneksi PDO (*PHP Data Objects*), dengan parameter *:username* yang kemudian diikat nilainya melalui *bindParam().* Setelah eksekusi perintah SQL, hasil pencarian disimpan dalam variabel *$user* dalam bentuk *array asosiatif*.

Jika data pengguna ditemukan *($user tidak bernilai null*) dan nilai *password* yang dimasukkan sama persis dengan *password* yang tersimpan dalam *database*, maka pengguna dianggap berhasil masuk dan sistem akan mengarahkan pengguna ke halaman *dashboard* melalui perintah *header("Location: ...").* Namun, jika autentikasi gagal, baik karena *username* tidak ditemukan atau *password* tidak sesuai, maka sistem akan menyimpan pesan kesalahan *'Invalid username or password*.' ke dalam variabel sesi *$_SESSION['error'].* Setelah itu, pengguna akan diarahkan kembali ke halaman *login*.

Sebagai bentuk penanganan kesalahan, skrip ini juga menggunakan blok *try-catch* untuk menangkap dan menampilkan pesan *error* dari objek *PDOException* apabila terjadi kegagalan koneksi atau eksekusi *query* SQL. Dengan demikian, pengguna akan tetap diarahkan ke halaman *login* apabila terjadi *error*.

```
<!-- form registrasi -->
<div class="register-container" id="register">
    <div class="top">
        <header>Sign Up</header>
    </div>
    <form action="http://localhost/TA/Website%20(1)/process_register.php" method="POST">
        <div class="two-forms">
            <div class="input-box">
                <input type="text" name="firstname" class="input-field" placeholder="Firstname" required>
                <i class="bx bx-user"></i>
            </div>
            <div class="input-box">
                <input type="text" name="lastname" class="input-field" placeholder="Lastname" required>
                <i class="bx bx-user"></i>
            </div>
        </div>
        <div class="input-box">
            <input type="text" name="no_induk" class="input-field" placeholder="No Induk" required>
            <i class="bx bx-id-card"></i>
        </div>
        <div class="input-box">
            <input type="text" name="username" class="input-field" placeholder="Username" required>
            <i class="bx bx-user-circle"></i>
        </div>
        <div class="input-box">
            <input type="password" name="password" class="input-field" placeholder="Password" required>
            <i class="bx bx-lock-alt"></i>
        </div>
        <div class="input-box">
            <input type="submit" class="submit" value="Register">
        </div>
        <div class="top">
            <span>Have an account? <a href="#" onclick="login()">Login</a></span>
        </div>
    </form>
</div>
```

Gambar 4.47 Struktur Form Registrasi

Formulir registrasi yang ditampilkan pada Gambar 4.47 disusun menggunakan elemen-elemen HTML. Secara umum, struktur form ini berada di dalam elemen * * dengan kelas *register-container* dan memiliki atribut *id="register",* yang memudahkan pemanggilan dinamis melalui *JavaScript* untuk keperluan pengalihan tampilan (*switching view*) antara form *login* dan *registrasi*.

Di dalam *form* tersebut, pengguna diminta untuk mengisi beberapa data penting yang dibutuhkan untuk membuat akun baru, antara lain nama depan (*firstname*), nama belakang (*lastname*), nomor induk (*no_induk*), *username*, dan *password*. Masing-masing *input* dibungkus dalam elemen * * yang memiliki kelas *input-box*, sehingga secara visual dan logika pemrograman dapat dikelompokkan dengan baik. *Input firstname* dan *lastname* dikelompokkan bersama dalam satu * * dengan kelas *two-forms*, yang menunjukkan bahwa keduanya ditampilkan secara berdampingan.

Setiap elemen *input* menggunakan tipe *text* kecuali untuk *password* yang menggunakan tipe *password*, yang berfungsi untuk menyembunyikan karakter saat diketik demi menjaga kerahasiaan informasi yang dimasukkan. Selain itu, setiap input memiliki atribut *name*, yang digunakan untuk mengidentifikasi nilai *input* saat dikirimkan ke *server* melalui metode *POST*, serta atribut *placeholder* sebagai petunjuk singkat mengenai data yang harus diisi oleh pengguna. Elemen <*i*> dari *library Boxicons* ditambahkan di dalam setiap *input-box* untuk memberikan ikon yang relevan secara visual, seperti ikon pengguna atau ikon kunci, yang memperkaya pengalaman antarmuka pengguna.

Formulir ini akan dikirimkan ke server melalui elemen <*form*> dengan atribut *action* mengarah ke *file process_register.php* dan *method="POST",* yang menunjukkan bahwa data akan dikirimkan secara tersembunyi melalui metode *POST* untuk alasan keamanan. Tombol *submit* bertipe *input* dengan kelas *submit* berfungsi untuk mengirimkan seluruh data isian pengguna ke *server* saat ditekan. Di bagian bawah *form*, terdapat pula teks navigasi tambahan yang memberikan tautan untuk kembali ke halaman login apabila pengguna telah memiliki akun, yang ditangani oleh fungsi *JavaScript bernama login()*.

```php
<?php
include 'db_connection.php';

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $firstname = htmlspecialchars($_POST['firstname']);
    $lastname = htmlspecialchars($_POST['lastname']);
    $no_induk = htmlspecialchars($_POST['no_induk']);
    $username = htmlspecialchars($_POST['username']);
    $password = $_POST['password'];  // Menyimpan password langsung tanpa hashing

    try {
        $sql = "INSERT INTO users (firstname, lastname, no_induk, username, password)
                VALUES (:firstname, :lastname, :no_induk, :username, :password)";
        $stmt = $conn->prepare($sql);
        $stmt->bindParam(':firstname', $firstname);
        $stmt->bindParam(':lastname', $lastname);
        $stmt->bindParam(':no_induk', $no_induk);
        $stmt->bindParam(':username', $username);
        $stmt->bindParam(':password', $password);

        $stmt->execute();

        // Redirect ke halaman login setelah registrasi berhasil
        header("Location: Login Page.html");
        exit;
    } catch (PDOException $e) {
        echo "Error: " . $e->getMessage();
    }
}
?>
```

Gambar 4.48 Kode Pada File process_register.php

Gambar 4.48 merupakan kode pada *file process_register.php.* Kode backend pada *file process_register.php* bertanggung jawab untuk menangani proses pendaftaran pengguna baru pada sistem. Proses ini dimulai dengan menyertakan (*include*) berkas *db_connection.php*, yang berfungsi untuk menghubungkan skrip PHP dengan basis data menggunakan objek koneksi *$conn* yang dikonfigurasi dengan *library* PDO (*PHP Data Objects*).

Selanjutnya, skrip ini memeriksa apakah permintaan (*request*) yang diterima berasal dari metode *POST*, yang menandakan bahwa data telah dikirim melalui formulir HTML. Pemeriksaan ini dilakukan menggunakan pernyataan kondisional *if ($_SERVER['REQUEST_METHOD'] === 'POST')*, yang merupakan praktik standar untuk menghindari pemrosesan skrip jika tidak ada data yang dikirimkan.

Jika kondisi tersebut terpenuhi, data yang dikirim melalui formulir yaitu nama depan (*firstname*), nama belakang (*lastname*), nomor induk (*no_induk*), nama pengguna (*username*), dan kata sandi (*password*) diambil dari *array $_POST*. Empat dari lima data tersebut disanitasi menggunakan fungsi *htmlspecialchars()*, yaitu *firstname*, *lastname*, *no_induk*, dan *username*. Proses filterisasi ini bertujuan untuk mengubah karakter-karakter khusus (seperti <, >, dan &) menjadi karakter HTML yang aman, sehingga dapat mencegah serangan injeksi HTML maupun XSS (*Cross-Site Scripting*) pada data *input* yang akan disimpan ke dalam *database* atau ditampilkan kembali di antarmuka pengguna.

Setelah data *input* berhasil disiapkan, sistem mencoba menjalankan perintah *SQL INSERT INTO users*, yang bertujuan untuk menyimpan seluruh informasi pengguna ke dalam tabel *users*. Penyisipan data ini dilakukan melalui metode yang aman, yaitu *prepared statement*, dengan parameter yang ditandai oleh tanda titik dua (misalnya *:firstname*). Objek *$stmt* yang merupakan hasil dari *$conn->prepare($sql)* digunakan untuk mempersiapkan perintah SQL tersebut, dan selanjutnya dilakukan *binding* terhadap setiap parameter dengan nilai variabel yang bersesuaian menggunakan metode *bindParam()*.

Jika seluruh parameter telah diikat dan perintah dieksekusi menggunakan *$stmt->execute()*, maka data pengguna secara resmi ditambahkan ke dalam *database*. Setelah proses penyimpanan berhasil, pengguna langsung diarahkan ke halaman *login* (*Login Page.html*) melalui perintah *header*("*Location: Login Page.html*"), yang menandakan bahwa registrasi telah selesai dan pengguna dapat melanjutkan proses masuk (*login*). Perintah *exit* dipanggil segera setelahnya untuk memastikan bahwa eksekusi skrip dihentikan dan tidak ada baris kode tambahan yang dijalankan secara tidak sengaja.

Sebagai langkah penanganan kesalahan, blok *try-catch* digunakan untuk menangkap kemungkinan pengecualian (*exception*) yang dapat muncul selama proses penyimpanan data, terutama yang bersifat terkait *database* (misalnya duplikasi *username* atau kegagalan koneksi). Jika terjadi kesalahan, sistem akan mencetak pesan *error* melalui perintah *echo*.

```
<script>
    function myMenuFunction() {
    var i = document.getElementById("navMenu");
    if(i.className === "nav-menu") {
        i.className += " responsive";
    } else {
        i.className = "nav-menu";
    }
    }
    function login() {
        document.getElementById("login").style.left = "4px";
        document.getElementById("register").style.right = "-520px";
        document.getElementById("loginBtn").className += " white-btn";
        document.getElementById("registerBtn").className = "btn";
    }
    function register() {
        document.getElementById("login").style.left = "-510px";
        document.getElementById("register").style.right = "5px";
        document.getElementById("loginBtn").className = "btn";
        document.getElementById("registerBtn").className += " white-btn";
    }
</script>
```

Gambar 4.49 Kode Fungsi myMenuFunction(), login(), dan register()

Kode JavaScript yang ditampilkan pada Gambar 4.49 terdiri dari tiga fungsi utama, yaitu *myMenuFunction(), login(), dan register().* Ketiga fungsi ini secara umum berperan dalam mengatur responsivitas serta transisi antarmuka halaman *login* dan *registrasi*.

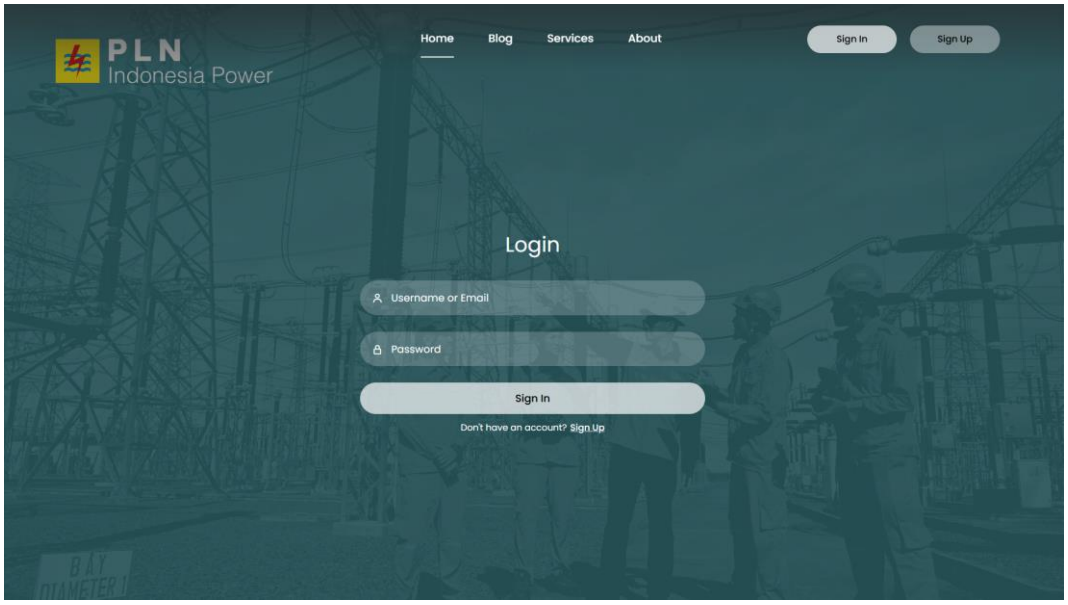
Fungsi *myMenuFunction()* digunakan untuk mengelola tampilan menu navigasi agar menjadi lebih responsif ketika diakses melalui perangkat dengan layar kecil, seperti ponsel pintar. Di dalam fungsi ini, objek elemen HTML yang memiliki *id="navMenu"* pertama-tama diambil melalui metode *document.getElementById("navMenu")* dan disimpan dalam variabel *<i>*. Kemudian, dilakukan pengecekan terhadap properti *className* dari elemen tersebut. Jika *className* saat ini adalah "*nav-menu*", maka kelas responsive akan ditambahkan ke dalamnya, sehingga nilai akhir dari *className* menjadi "*nav-menu responsive*". Penambahan kelas ini umumnya akan mengaktifkan aturan CSS tambahan yang mengubah tampilan menu menjadi lebih sesuai untuk perangkat

mobile. Sebaliknya, jika kelas responsive telah ditambahkan sebelumnya, maka kondisi else akan dijalankan untuk mengembalikan *className* ke nilai awal "*nav-menu*", sehingga menu kembali ke bentuk *default* untuk tampilan desktop.

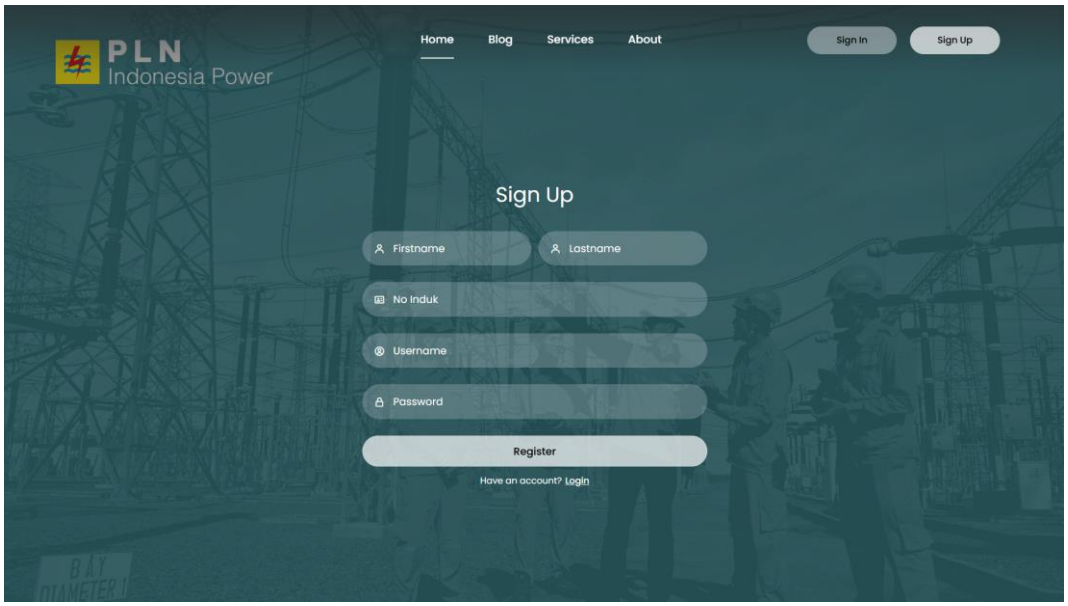
Fungsi berikutnya, yaitu *login(),* berperan dalam menampilkan antarmuka formulir *login* dan menyembunyikan formulir registrasi. Pada fungsi ini, elemen dengan *id="login"* diatur properti gaya CSS-nya agar bergeser ke kiri (*left = "4px"*), sehingga tampak di area tampilan. Sementara itu, elemen dengan *id="register"* digeser ke kanan jauh di luar layar (*right = "-520px"*) sehingga tidak terlihat oleh pengguna. Fungsi ini juga melakukan manipulasi kelas (*class*) untuk dua tombol, yaitu tombol *login (id="loginBtn")* dan tombol registrasi (*id="registerBtn"*). Pada tombol *login*, ditambahkan kelas *white-btn*, yang secara visual kemungkinan besar memberikan penegasan bahwa tombol tersebut sedang aktif. Sementara itu, pada tombol registrasi, kelasnya dikembalikan hanya menjadi "*btn*", yang berarti tampilannya dinetralkan.

Adapun fungsi *register()* memiliki logika yang berkebalikan dengan fungsi *login().* Fungsi ini memindahkan posisi formulir *login* keluar dari tampilan dengan mengatur nilai *left* menjadi "-510px", serta menampilkan formulir registrasi dengan memindahkannya ke posisi terlihat (*right* = "5px"). Dalam hal gaya tombol, kelas *white-btn* ditambahkan pada tombol registrasi untuk menonjolkan tombol tersebut, sedangkan kelas tombol *login* dikembalikan ke "*btn*" tanpa penekanan visual tambahan.

Dengan demikian, ketiga fungsi ini berkontribusi terhadap peningkatan pengalaman pengguna (*user experience*) melalui pengendalian tampilan dinamis tanpa harus memuat ulang halaman. Seluruh interaksi ini berjalan secara *client-side* menggunakan mekanisme manipulasi DOM (*Document Object Model*) yang efisien. Walaupun kode ini telah mencukupi untuk kebutuhan dasar antarmuka interaktif, namun dalam pengembangan lanjutan, penggunaan *library* seperti *jQuery* atau *framework* modern seperti *React* atau *Vue* dapat memberikan fleksibilitas dan efisiensi lebih tinggi, terutama dalam pengelolaan antarmuka yang kompleks. Untuk meninjau hasil kode yang dibuat dapat melihat Gambar 4.50 dan Gambar 4.51 yang menampilkan *Login Page* dan *Register Page*.



Gambar 4.50 Login Page Website



Gambar 4.51 Register Page Website

4.2.3.2 Pembuatan Program Dashboard Page pada Website

Halaman *dashboard* pada sistem monitoring dispenser *earplug* otomatis dirancang untuk menampilkan data secara informatif dan interaktif kepada pengguna, khususnya administrator. Komponen *frontend* dibangun menggunakan

struktur HTML yang disusun secara responsif dengan bantuan sistem grid berbasis *class* seperti *row* dan *col-md-x*, yang memungkinkan tampilan tetap rapi di berbagai ukuran layar.

```
<!-- Dashboard Content -->
<div id="dashboardContent" class="content-section">
    <div class="row mb-4">
        <h3>Dashboard</h3>
        <!-- Input Tanggal dan Tombol untuk Mengubah Tanggal -->
        <div class="form-group">
            <label for="searchDate">Pilih Tanggal</label>
            <input type="date" id="searchDate" class="form-control" />
            <button id="filterButton" class="btn-clear ml-2">Tampilkan</button>
        </div>

        <p></p>
        <div class="col-md-6">
            <div class="card text-center">
                <div class="card-body">
                    <h5>Total Earplug Available</h5>
                    <h2 id="totalEarplugAvailable">Loading...</h2>
                    <div id="earplugWarningIcon" style="margin-top: 10px;"></div>
                </div>
            </div>
        </div>

        <div class="col-md-6">
            <div class="card text-center">
                <div class="card-body">
                    <h5>Total Used Today</h5>
                    <h2 id="totalUsedToday">Loading...</h2>
                </div>
            </div>
        </div>

        <!-- Fungsi Chart di Dashboard -->
        <div class="row mt-4">

        <!-- Chart Mingguan  -->
        <div class="row mt-4">
```

```
        <!-- Chart Mingguan  -->
        <div class="row mt-4">
            <div class="col-12">
                <h5>History Penggunaan Mingguan</h5>
                <canvas id="weeklyUsageChart" width="400" height="100"></canvas>
            </div>
        </div>

        <!-- Chart Bulanan dan Tahunan -->
        <div class="row mt-4">
            <!-- Chart Bulanan -->
            <div class="col-md-6">
                <h5>History Penggunaan Bulanan</h5>
                <canvas id="monthlyUsageChart" width="400" height="200"></canvas>
            </div>

            <!-- Chart Tahunan -->
            <div class="col-md-6">
                <h5>History Penggunaan Tahunan</h5>
                <canvas id="yearlyUsageChart" width="400" height="200"></canvas>
            </div>
        </div>
        </div>
    </div>
</div>
```

Gambar 4.52 Kode HTML untuk Dashboard Content

Gambar 4.52 menunjukkan program HTML untuk *dashboard content*. Pada bagian awal halaman, terdapat elemen judul utama berupa *<h3>* dengan label "*Dashboard*" sebagai penanda konteks halaman. Di bawahnya, terdapat sebuah *form* yang terdiri atas *input* bertipe *date* dengan *id="searchDate"* serta tombol bertuliskan "Tampilkan" yang memiliki *id="filterButton"*. *Form* ini berfungsi sebagai sarana filter data berdasarkan tanggal tertentu, sehingga pengguna dapat menampilkan informasi penggunaan earplug sesuai dengan tanggal yang diinginkan.

Selanjutnya, terdapat dua buah *card* yang masing-masing menampilkan data penting terkait ketersediaan dan penggunaan *earplug*. *Card* pertama memuat informasi "*Total Earplug Available*" dengan nilai yang akan ditampilkan secara dinamis pada elemen *id="totalEarplugAvailable"*. Selain itu, terdapat elemen *div* tambahan dengan *id="earplugWarningIcon"* yang disiapkan untuk menampilkan peringatan visual apabila jumlah *earplug* berada dalam kondisi minimum. *Card*

kedua menampilkan data “Total Used Today” dengan nilai yang ditampilkan pada *id="totalUsedToday"* sebagai informasi pemakaian *earplug* pada hari yang sama.

Di bagian bawah halaman, ditampilkan visualisasi data dalam bentuk grafik (*chart*) menggunakan elemen <*canvas*> yang akan diisi melalui pustaka JavaScript tertentu. Terdapat tiga kategori grafik, yaitu grafik mingguan (*id="weeklyUsageChart"*), bulanan (id="monthlyUsageChart"), dan tahunan (*id="yearlyUsageChart"*). Ketiga grafik ini digunakan untuk memvisualisasikan riwayat penggunaan *earplug* dalam rentang waktu yang berbeda, sehingga memudahkan pengguna dalam melakukan analisis data pemakaian.

```css
/* Style Dashboard */
body {
    font-family: 'Poppins', sans-serif;
    background-color: #f8f9fa;
    margin: 0;
    padding: 0;
}
```

Gambar 4.53 Kode Dasar Style Body

Penataan tampilan antarmuka halaman *Dashboard* dilakukan dengan menggunakan *Cascading Style Sheets* (CSS). Tujuannya adalah untuk menciptakan antarmuka pengguna yang bersih, responsif, dan mudah dipahami oleh pengguna. Gambar 4.53 merupakan kode dasar *style body* seperti pengaturan jenis huruf dengan *font-family: 'Poppins', sans-serif;* untuk memberikan kesan modern dan profesional, serta *background-color: #f8f9fa;* pada elemen *body* untuk menghasilkan latar belakang berwarna abu-abu terang yang nyaman dipandang mata.

```css
/* Main Content */
.main-content {
    margin-left: 250px;
    padding: 2rem;
    overflow-x: hidden;
    position: relative;
    transition: margin-left 0.3s ease;
}

.main-content h3 {
    font-weight: bold;
    color: #337780;
}

.card {
    border: none;
    box-shadow: 0 6px 12px rgba(0, 0, 0, 0.1);
    border-radius: 10px;
    background-color: #ffffff;
    margin-bottom: 20px;
}

.card h5 {
    font-size: 1.2rem;
    color: #555;
}

.card h2 {
    font-size: 2rem;
    font-weight: bold;
    color: #337780;
}

.form-control {
    border-radius: 5px;
}
```

Gambar 4.54 Kode Untuk Mengatur Style Elemen Dan Informasi Utama

Gambar 4.54 merupakan kode untuk mengatur *style* elemen dan informasi utama. Elemen utama halaman yang memuat konten, yaitu *main-content*, diberikan pengaturan *margin* kiri sebesar 250 piksel (*margin-left: 250px*) agar tidak bertabrakan dengan elemen *sidebar*. Selain itu, diberikan *padding* sebesar 2rem untuk memberikan ruang antar elemen dalam konten, serta *overflow-x: hidden* untuk mencegah kemunculan *scroll* horizontal yang tidak diinginkan.

Komponen informasi utama ditampilkan dalam elemen kartu (*card*). Elemen ini dirancang menggunakan kelas *.card* dengan *box-shadow* dan *border-radius* untuk memberikan efek bayangan dan sudut membulat. Judul di dalam kartu menggunakan kelas *.card h5* dengan warna abu-abu dan ukuran sedang, sedangkan

nilai statistik utama seperti jumlah *earplug* menggunakan kelas *.card h2* dengan ukuran lebih besar dan kode warna #337780.

```css
.warning {
    background-color: #e90202 !important;
    animation: blink 1s infinite;
    color: black;
}

.warning h2,
.warning h5 {
    color: white !important;
}


@keyframes blink {
    0% { opacity: 1; }
    50% { opacity: 0.5; }
    100% { opacity: 1; }
}
```

Gambar 4.55 Kode CSS Untuk Kelas .warning

Untuk memberikan peringatan visual apabila jumlah *earplug* menipis, digunakan kelas *.warning* seperti pada Gambar 4.55 yang menerapkan warna latar belakang merah (#e90202) dengan efek animasi berkedip menggunakan properti *animation: blink 1s infinite*. Warna teks pada mode peringatan diubah menjadi putih agar tetap kontras dan mudah terbaca.

```css
.form-control {
    border-radius: 5px;
}
```

Gambar 4.56 Kode CSS Untuk Kelas .form-control

Elemen formulir seperti *input* tanggal *(<input type="date">)* dikustomisasi dengan kelas *.form-control* seperti pada Gambar 4.56, yang memiliki sudut membulat (*border-radius*). Selain itu, tombol-tombol seperti “Tampilkan” menggunakan kelas *.btn-clear* dan *.btn-submit* dengan warna dasar hijau kebiruan serta efek perubahan

warna saat di-*hover* untuk meningkatkan interaktivitas. Gambar 4.57 menunjukkan program yang digunakan untuk *style btn-clear* dan *btn-submit.*

```css
/* Button Clear and Submit */
.btn-clear, .btn-submit {
    background-color: #337780;
    color: white;
    font-weight: bold;
    border-radius: 5px;
    padding: 10px 15px;
    border: none;
    transition: background-color 0.3s ease;
}

.btn-clear:hover, .btn-submit:hover {
    background-color: #285d63;
}
```

Gambar 4.57 Kode CSS Untuk Style btn-clear dan btn-submit

Terakhir, kelas .row diberikan *margin-top* tambahan untuk memberikan jarak antar grafik (*chart*) sehingga susunan informasi terlihat lebih rapi dan tidak saling bertumpukan. Gambar 4.58 menunjukkan *style* yang digunakan untuk *class row*.

```css
.row {
    margin-top: 20px; /* Menambahkan jarak atas pada chart */
}
```

Gambar 4.58 Kode CSS Untuk Kelas .row

Untuk membantu eksekusi data dengan lebih baik diperlukan kode berbasis JavaScript. Gambar 4.59 menunjukkan kode yang digunakan untuk melakukan filter tanggal pada *dashboard page*. Bagian kode JavaScript tersebut digunakan untuk menambahkan fungsionalitas penyaringan data berdasarkan tanggal pada halaman *dashboard*. Fungsionalitas ini diimplementasikan dengan menggunakan metode *event listener* yang dipasang pada tombol dengan atribut *id="filterButton"*. Ketika tombol diklik, fungsi akan dijalankan untuk mengambil nilai tanggal yang

dipilih oleh pengguna melalui elemen *input* bertipe tanggal yang memiliki atribut *id="searchDate"*.

```
// Menambahkan fungsionalitas pencarian berdasarkan tanggal
document.getElementById('filterButton').addEventListener('click', function () {
    // Ambil tanggal yang dipilih
    const selectedDate = document.getElementById('searchDate').value;

    // Cek apakah tanggal valid
    if (selectedDate) {
        // Ambil data total penggunaan hari ini berdasarkan tanggal yang dipilih
        fetch(`total_used_today.php?searchDate=${selectedDate}`)
            .then(response => {
                // Periksa apakah response berhasil
                if (!response.ok) {
                    throw new Error('Failed to load total used today for selected date');
                }
                return response.json();
            })
            .then(data => {
                if (data.usedToday !== undefined) {
                    document.getElementById('totalUsedToday').innerText = data.usedToday;
                } else {
                    console.error('Invalid data for total used today:', data);
                    document.getElementById('totalUsedToday').innerText = '0'; // Jika tidak ada data
                }
            })
            .catch(error => console.error('Error loading total used today:', error));
    } else {
        alert('Pilih tanggal terlebih dahulu!');
    }
});
```

Gambar 4.59 Kode Filter Tanggal Pada Dashboard Page

Setelah nilai tanggal diperoleh, dilakukan pemeriksaan untuk memastikan bahwa pengguna telah memilih tanggal. Jika tidak ada tanggal yang dipilih, sistem akan menampilkan peringatan dalam bentuk *alert* untuk meminta pengguna memilih tanggal terlebih dahulu. Namun, apabila tanggal telah dipilih, maka proses dilanjutkan dengan melakukan permintaan data ke sisi *backend* menggunakan fungsi *fetch*. Permintaan ini ditujukan ke berkas *total_used_today.php* dengan menyertakan parameter *searchDate* yang nilainya sesuai dengan tanggal yang dipilih pengguna.

Data yang diterima dari *backend* diharapkan dalam format *JSON*. Oleh karena itu, respons dari permintaan tersebut diubah ke dalam bentuk *JSON* untuk kemudian diproses lebih lanjut. Jika data yang diterima mengandung nilai properti usedToday, maka nilai tersebut akan ditampilkan pada elemen HTML dengan *id="totalUsedToday"*. Sebaliknya, jika data tidak valid atau tidak memuat

informasi yang diharapkan, maka sistem akan menampilkan nilai *default* berupa nol (0) dan mencetak pesan kesalahan ke *console* pengembang untuk keperluan *debugging*.

```
//Fungsi Untuk memperbarui Grafik Mingguan
let weeklyChartInstance = null;

function updateWeeklyUsageChart(weeklyUsageData) {
    const sortedDates = Object.keys(weeklyUsageData).sort(); // Urutkan tanggal

    const counts = sortedDates.map(date => weeklyUsageData[date] || 0);

    const ctx = document.getElementById('weeklyUsageChart').getContext('2d');
```

Gambar 4.60 Kode Visualisasi Data Dalam Grafik Mingguan

Setelah sebelumnya dibahas mengenai mekanisme *filtering* data berdasarkan tanggal tertentu, bagian kode pada Gambar 4.60 berfokus pada visualisasi data dalam bentuk grafik mingguan. Visualisasi ini bertujuan untuk memberikan gambaran mengenai tren penggunaan harian selama tujuh hari terakhir, sehingga pengguna dapat memantau perubahan atau kecenderungan penggunaan secara lebih intuitif melalui grafik batang (*bar chart*).

Fungsi utama yang bertanggung jawab terhadap pembaruan grafik adalah *updateWeeklyUsageChart(weeklyUsageData)* yang ditunjukkan pada Gambar 4.60. Fungsi ini pertama-tama mengurutkan data berdasarkan tanggal menggunakan *Object.keys(weeklyUsageData).sort(),* sehingga grafik menampilkan data secara kronologis. Selanjutnya, nilai penggunaan per tanggal dimasukkan ke dalam *array counts* yang kemudian digunakan sebagai data utama dalam grafik.

```
if (weeklyChartInstance) {
    weeklyChartInstance.destroy();
}

weeklyChartInstance = new Chart(ctx, {
    type: 'bar',
    data: {
        labels: sortedDates, // Tanggal sebagai label
        datasets: [{
            label: 'Penggunaan Harian',
            data: counts,
            backgroundColor: 'rgba(255, 159, 64, 0.2)',
            borderColor: 'rgb(255, 159, 64)',
            borderWidth: 2,
            tension: 0.3,
            fill: true,
        }]
    },
    options: {
        responsive: true,
        scales: {
            x: {
                title: {
                    display: true,
                    text: 'Tanggal'
                }
            },
            y: {
                title: {
                    display: true,
                    text: 'Jumlah Penggunaan'
                },
                beginAtZero: true
            }
        }
    }
```

Gambar 4.61 Kode Untuk Pembuatan Chart Baru Pada Grafik Mingguan

Elemen *canvas* HTML dengan *id="weeklyUsageChart"* menjadi tempat grafik ditampilkan. Jika sebelumnya telah terdapat instansi grafik (*chart instance*), maka grafik lama dihilangkan terlebih dahulu menggunakan *destroy()* untuk mencegah duplikasi grafik. Setelah itu, dibuat objek *Chart* baru menggunakan pustaka *Chart.js*, dengan tipe grafik batang. Grafik ini menampilkan tanggal sebagai label sumbu-x, dan jumlah penggunaan sebagai nilai pada sumbu-y. Opsi visual seperti warna batang, garis batas, ketebalan, dan efek *tension* juga disesuaikan agar grafik tampil informatif dan menarik. Gambar 4.61 menunjukkan proses pembuatan *chart* baru pada grafik mingguan.

```
//fetch usage history data untuk grafik mingguan
function fetchWeeklyUsageData() {
    fetch('get_usage_history.php')
        .then(response => response.json())
        .then(data => {
            const weeklyUsageData = {};
            const today = new Date();
            const sevenDaysAgo = new Date();
            sevenDaysAgo.setDate(today.getDate() - 7); // 6 hari sebelumnya + hari ini = 7 hari

            data.forEach(row => {
                const date = new Date(row.date);
                const formattedDate = date.toISOString().split('T')[0]; // Format YYYY-MM-DD

                // Hanya hitung data yang status-nya BUKAN 'limit_exceeded'
                if (
                    date >= sevenDaysAgo &&
                    date <= today &&
                    row.status !== 'limit_exceeded'
                ) {
                    if (!weeklyUsageData[formattedDate]) {
                        weeklyUsageData[formattedDate] = 0;
                    }

                    // Pastikan nilai digunakan adalah angka
                    weeklyUsageData[formattedDate] += parseInt(row.used_quantity);
                }
            });

            updateWeeklyUsageChart(weeklyUsageData);
        })
        .catch(error => {
            console.error('Error fetching weekly usage data:', error);
        });
}
```

Gambar 4.62 Kode Fetch Grafik Mingguan

Untuk memperoleh data yang digunakan dalam grafik, fungsi *fetchWeeklyUsageData()* dipanggil. Fungsi ini melakukan permintaan data ke *file backend get_usage_history.php*, yang mengembalikan riwayat penggunaan dalam format *JSON*. Setiap entri data yang diterima akan diperiksa: hanya data dalam rentang tujuh hari terakhir dan yang tidak memiliki status *'limit_exceeded'* yang akan dihitung. Tanggal akan diformat ke dalam format standar *ISO* (YYYY-MM-DD), dan data penggunaan per tanggal dijumlahkan ke dalam objek *weeklyUsageData*. Gambar 4.62 menunjukkan kode *fetch* grafik mingguan.

Setelah seluruh data dihimpun, fungsi *updateWeeklyUsageChart()* dipanggil untuk memperbarui grafik. Selain itu, untuk menjaga agar grafik selalu menampilkan data terkini, fungsi *fetchWeeklyUsageData()* dijalankan secara

otomatis setiap 10 detik menggunakan *setInterval()*, dan juga dipanggil sekali ketika halaman pertama kali dimuat. Gambar 4.63 menunjukkan kode *setInterval()* yang digunakan untuk mendapatkan data terbaru setiap 10 detik.

```
setInterval(() => {
    fetchWeeklyUsageData(); // Update setiap 10 detik
}, 10000);

fetchWeeklyUsageData(); // Panggil saat halaman dimuat
```

Gambar 4.63 Kode setInterval()

Dengan mekanisme ini, sistem dapat menyajikan pembaruan grafik mingguan secara *real-time* tanpa perlu *refresh* manual, yang meningkatkan pengalaman pengguna dalam memantau penggunaan secara dinamis dan efisien.

```
// Fungsi untuk memperbarui grafik bulanan penggunaan (mencakup beberapa tahun)
function updateMonthlyUsageChart(monthlyUsageData) {
    // Buat array dari kunci bulan dan tahun
    const monthsOrder = Object.keys(monthlyUsageData);

    // Urutkan bulan berdasarkan tahun dan bulan dengan format YYYY-MM
    monthsOrder.sort((a, b) => {
        const [monthA, yearA] = a.split(' ');  // Pisahkan bulan dan tahun
        const [monthB, yearB] = b.split(' ');

        // Map bulan ke angka untuk memudahkan perbandingan
        const monthMap = { 'January': 1, 'February': 2, 'March': 3, 'April': 4, 'May': 5,
        'June': 6, 'July': 7, 'August': 8, 'September': 9, 'October': 10, 'November': 11, 'December': 12 };

        // Perbandingan antara tahun dan bulan
        if (yearA === yearB) {
            return monthMap[monthA] - monthMap[monthB]; // Jika tahun sama, urutkan berdasarkan bulan
        } else {
            return yearA - yearB; // Urutkan berdasarkan tahun
        }
    });

    const counts = monthsOrder.map(monthYear => monthlyUsageData[monthYear] || 0); // Ambil data untuk setiap bulan dengan tahun

    const ctx = document.getElementById('monthlyUsageChart').getContext('2d');
```

Gambar 4.64 Kode Program Untuk Visualisasi Isi Grafik Bulanan

Setelah pembahasan mengenai grafik mingguan, sistem juga menyediakan visualisasi data dalam bentuk grafik bulanan untuk memantau tren penggunaan dalam jangka waktu yang lebih panjang. Visualisasi bulanan ini sangat bermanfaat untuk melakukan analisis periodik yang mencakup beberapa bulan bahkan tahun. Seperti halnya grafik mingguan, grafik ini juga menggunakan pustaka *Chart.js*,

namun terdapat beberapa perbedaan signifikan dalam cara pengolahan dan penyajian datanya. Gambar 4.64 menunjukkan program untuk visualisasi isi grafik bulanan.

Fungsi *updateMonthlyUsageChart(monthlyUsageData)* bertugas untuk menampilkan grafik batang penggunaan bulanan. Salah satu perbedaan utama dari grafik mingguan terletak pada format label sumbu-x, di mana data ditampilkan dalam format gabungan bulan dan tahun, misalnya "January 2024", "February 2024", dan seterusnya. Untuk memastikan urutan data yang logis secara kronologis, dilakukan proses pengurutan secara manual berdasarkan urutan bulan dan tahun. Hal ini dilakukan dengan terlebih dahulu memetakan nama bulan ke angka melalui objek *monthMap*, agar dapat dibandingkan dan diurutkan dengan benar.

```
// Fetch History Data untuk Grafik Bulanan
function fetchUsageHistoryData() {
    fetch('get_usage_history.php')
        .then(response => response.json())
        .then(data => {
            const monthlyUsageData = {};

            // Proses data ke dalam format bulanan
            data.forEach(row => {
                const date = new Date(row.date);
                const month = date.toLocaleString('default', { month: 'long' });
                const year = date.getFullYear();
                const monthYearKey = `${month} ${year}`;

                // Hanya hitung data yang status-nya BUKAN 'limit_exceeded'
                if (row.status !== 'limit_exceeded') {
                    if (!monthlyUsageData[monthYearKey]) {
                        monthlyUsageData[monthYearKey] = 0;
                    }
                    monthlyUsageData[monthYearKey] += parseInt(row.used_quantity);
                }
            });

            // Perbarui grafik setelah data diterima
            updateMonthlyUsageChart(monthlyUsageData);
        })
        .catch(error => {
            console.error('Error fetching usage data:', error);
        });
}
```

Gambar 4.65 Kode Fetch Untuk Grafik Bulanan

Gambar 4.65 menunjukkan kode *fetch* untuk grafik bulanan. Berbeda dengan grafik mingguan yang hanya mencakup tujuh hari terakhir, grafik bulanan memuat data secara agregat per bulan, tanpa batasan waktu tertentu. Proses pengambilan data dilakukan melalui fungsi *fetchUsageHistoryData()*, yang mengambil semua data riwayat penggunaan dari *file backend get_usage_history.php*. Setiap entri data dikonversi ke dalam format bulanan menggunakan *toLocaleString()* untuk mengambil nama bulan dan *getFullYear()* untuk mengambil tahunnya. Gabungan dari keduanya membentuk *key* dengan format "Bulan Tahun" (misalnya "April 2025"), yang digunakan sebagai pengelompok data.

Sebagaimana grafik mingguan, hanya data dengan status yang tidak bernilai *'limit_exceeded'* yang diproses. Nilai penggunaan dari entri-entri ini kemudian dijumlahkan ke dalam objek m*onthlyUsageData*, sebelum akhirnya diproses lebih lanjut oleh *updateMonthlyUsageChart()* untuk diperbarui dalam tampilan grafik.

```
if (chartInstance) {
    chartInstance.destroy();
}

chartInstance = new Chart(ctx, {
    type: 'bar',
    data: {
        labels: monthsOrder, // Tampilkan bulan dengan tahun sebagai label
        datasets: [{
            label: 'Penggunaan per Bulan',
            data: counts,
            backgroundColor: 'rgba(75, 192, 192, 0.2)',
            borderColor: 'rgb(75, 192, 192)',
            borderWidth: 1,
            fill: true,
        }]
    },
    options: {
        responsive: true,
        scales: {
            x: {
                title: {
                    display: true,
                    text: 'Bulan'
                }
            },
            y: {
                title: {
                    display: true,
                    text: 'Jumlah Penggunaan'
                },
                beginAtZero: true
            }
        }
    }
```

Gambar 4.66 Kode Pengaturan Visual untuk Grafik Bulanan

Gambar 4.66 menujukkan proses pembuatan tampilan *chart* bulanan. Tampilan visual grafik bulanan menggunakan warna latar dan garis batas yang berbeda dari grafik mingguan, yakni warna biru kehijauan dengan kode warna *rgba(75, 192, 192, 0.2)* dan *rgb(75, 192, 192)*, yang secara visual membedakannya dalam konteks antarmuka. Selain itu, pemanggilan fungsi *fetchUsageHistoryData()* juga dijalankan secara berkala setiap 10 detik melalui *setInterval(),* memungkinkan grafik untuk diperbarui secara *real-time* dengan data terbaru.

Dengan perbedaan format label, rentang waktu, dan cara agregasi data, grafik bulanan ini memberikan dimensi tambahan dalam memahami pola penggunaan secara historis dalam jangka panjang, melengkapi fungsi pemantauan jangka pendek yang disediakan oleh grafik mingguan.

```
// Fetch Data History untuk Grafik Tahunan
function fetchAnnualUsageData() {
    fetch('get_usage_history.php')
        .then(response => response.json())
        .then(data => {
            const annualUsageData = {};

            // Proses data ke dalam format tahunan
            data.forEach(row => {
                const date = new Date(row.date);
                const year = date.getFullYear(); // Ambil tahun dari data

                // Hanya hitung data yang status-nya BUKAN 'limit_exceeded'
                if (row.status !== 'limit_exceeded') {
                    if (!annualUsageData[year]) {
                        annualUsageData[year] = 0;
                    }
                    annualUsageData[year] += parseInt(row.used_quantity);
                }
            });

            // Perbarui grafik setelah data diterima
            updateAnnualUsageChart(annualUsageData);
        })
        .catch(error => {
            console.error('Error fetching annual usage data:', error);
        });
}
```

Gambar 4.67 Kode Fetch Untuk Grafik Tahunan

Gambar 4.67 menunjukkan kode *fetch* untuk grafik tahunan. Tidak seperti grafik mingguan yang mencakup tujuh hari terakhir, dan grafik bulanan yang mengelompokkan data berdasarkan kombinasi bulan dan tahun, grafik tahunan menyajikan data penggunaan dalam cakupan waktu yang lebih luas, yaitu berdasarkan tahun kalender. Proses pengambilan data dilakukan melalui fungsi *fetchAnnualUsageData(),* yang mengambil seluruh data historis dari

get_usage_history.php, kemudian memproses setiap entri dengan menggunakan *getFullYear()* untuk mendapatkan nilai tahunnya. Nilai tahun tersebut digunakan sebagai key dalam objek *annualUsageData*, dan hanya data dengan status yang tidak bernilai 'limit_exceeded' yang dihitung. Nilai penggunaan dari entri-entri yang valid ini dijumlahkan berdasarkan tahun, sebelum dikirim ke fungsi *updateAnnualUsageChart()* untuk diperbarui ke dalam tampilan grafik.

Gambar 4.67 juga menunjukkan proses pembuatan tampilan *chart* tahunan. Grafik ini menggunakan visualisasi batang (*bar chart*) dengan skema warna ungu (rgba(153, 102, 255, 0.2) dan rgb(153, 102, 255)) untuk membedakannya dari grafik mingguan dan bulanan. Pemanggilan fungsi *fetchAnnualUsageData()* dijalankan secara berkala setiap 10 detik melalui *setInterval(),* memastikan grafik selalu menampilkan data terkini secara *real-time* tanpa memerlukan *refresh* manual.

Dengan cakupan waktu tahunan dan pemrosesan data berbasis rentang per tahun, grafik ini memberikan gambaran makro atas pola penggunaan dari waktu ke waktu. Kombinasi polling otomatis dan pengurutan label secara numerik *(sort((a, b) => a - b)*) memastikan grafik tetap relevan dan mudah dibaca dalam konteks pemantauan jangka panjang, melengkapi grafik mingguan yang bersifat harian dan bulanan yang bersifat menengah.

```
// Fungsi untuk memperbarui total penggunaan hari ini
function updateTotalUsedToday() {
    const today = new Date().toISOString().split('T')[0]; // Format YYYY-MM-DD
    fetch(`total_used_today.php?searchDate=${today}`)
        .then(response => response.json())
        .then(data => {
            document.getElementById('totalUsedToday').textContent = data.usedToday || 0;
        })
        .catch(error => console.error('Error fetching total used today:', error));
}
```

Gambar 4.68 Fungsi updateTotalUsedToday()

Setelah membahas representasi data historis dalam cakupan mingguan, bulanan, dan tahunan melalui visualisasi grafik, kini perhatian dialihkan pada penyajian informasi penggunaan terkini dalam skala harian. Elemen ini digunakan untuk memperbarui informasi pada *card* yang menunjukkan jumlah penggunaan *earplug* harian.

Kode pada Gambar 4.68 menunjukkan mekanisme pembaruan total penggunaan pada hari ini (*total used today*) melalui fungsi *updateTotalUsedToday()*. Fungsi ini bekerja dengan terlebih dahulu memperoleh tanggal saat ini dalam format standar ISO (*YYYY-MM-DD*) menggunakan *toISOString().split('T')[0],* sehingga memastikan kompatibilitas dengan format tanggal pada sisi *backend*. Nilai tanggal tersebut kemudian dikirim sebagai parameter pencarian melalui permintaan *fetch* ke skrip *total_used_today.php*. Setelah data berhasil diterima dalam bentuk *JSON*, nilai *usedToday* ditampilkan langsung pada elemen HTML dengan ID *totalUsedToday*. Apabila data tidak tersedia atau bernilai *null*, fungsi ini akan secara *default* menampilkan angka nol untuk mencegah tampilan kosong yang dapat membingungkan pengguna. Untuk mengantisipasi kemungkinan kegagalan dalam pengambilan data, telah disediakan blok *catch* yang mencetak pesan kesalahan ke konsol peramban sebagai bagian dari praktik penanganan kesalahan yang baik.

Dengan pendekatan ini, pembaruan data dilakukan secara *realtime* dan bersifat ringan, karena hanya mencakup satu nilai agregat harian. Fungsionalitas ini melengkapi informasi historis yang disediakan oleh grafik mingguan, bulanan, dan tahunan, serta memberikan konteks harian yang esensial dalam mendukung pengambilan keputusan atau pemantauan aktivitas sistem secara langsung.

```
// Fungsi untuk memperbarui total earplug tersedia
function updateTotalEarplugAvailable() {
    fetch('total_earplug.php')
        .then(response => response.json())
        .then(data => {
            const total = data.availableEarplug || 0;
            const displayElement = document.getElementById('totalEarplugAvailable');
            const cardElement = displayElement.closest('.card');
            const warningIconElement = document.getElementById('earplugWarningIcon');

            displayElement.textContent = total;

            if (total < 10) {
                cardElement.classList.add('warning');

                // Tambahkan ikon warning jika belum ada
                if (!warningIconElement.innerHTML.trim()) {
                    warningIconElement.innerHTML = `
                        <i class="bi bi-exclamation-triangle-fill"
                        style="color: yellow; font-size: 28px;"></i>
                    `;
                }
            } else {
                cardElement.classList.remove('warning');
                warningIconElement.innerHTML = ''; // hapus ikon kalau tidak perlu warning
            }
        })
        .catch(error => console.error('Error fetching total earplug available:', error));
}
```

Gambar 4.69 Fungsi updateTotalEarplugAvailable()

Setelah membahas fungsi *updateTotalUsedToday()* yang menampilkan jumlah penggunaan *earplug* pada hari yang sama, sistem juga menyediakan fitur lain yang menampilkan jumlah *earplug* yang masih tersedia. Informasi ini ditampilkan pada bagian *dashboard* untuk memberikan gambaran langsung mengenai ketersediaan alat pelindung yang siap digunakan.

Gambar 4.69 menunjukkan kode yang digunakan untuk memperbarui nilai *total earplug tersedia*. Fungsi *updateTotalEarplugAvailable()* bertugas mengambil data dari *file backend* total_earplug.php. Data dikembalikan dalam format *JSON* dan nilai *availableEarplug* ditampilkan ke elemen HTML dengan ID *totalEarplugAvailable*. Jika data tidak tersedia, maka nilai yang ditampilkan akan *default* ke nol.

Selain menampilkan angka, fungsi ini juga menyertakan pemeriksaan kondisi untuk menentukan apakah jumlah *earplug* tersedia berada di bawah ambang batas tertentu. Jika jumlahnya kurang dari 10, maka elemen *card* akan diberi kelas

tambahan *warning*, dan ikon peringatan akan ditampilkan pada elemen dengan ID *earplugWarningIcon*. Sebaliknya, jika jumlah mencukupi, kelas peringatan dan ikon akan dihapus untuk menjaga tampilan tetap informatif.

Dengan demikian, fungsi ini mendukung pemantauan stok secara *real-time* dan membantu pengguna dalam memastikan bahwa alat pelindung diri tersedia dalam jumlah yang memadai.

```php
<?php
// Include file koneksi database
include('dashboard_connection.php');

// Ambil parameter pencarian tanggal jika ada
$searchDate = isset($_GET['searchDate']) ? $_GET['searchDate'] : '';

// Query untuk mengambil data history usage
$query = "SELECT
            name,
            no_induk,
            DATE_FORMAT(date, '%Y-%m-%d') AS date,
            time,
            rfid_tag,
            used_quantity,
            department,
            status
            FROM history";

// Jika ada parameter pencarian tanggal, tambahkan kondisi WHERE
if ($searchDate) {
    $query .= " WHERE DATE(date) = :searchDate";
}

$query .= " ORDER BY date DESC, time DESC";

try {
    // Pastikan header dikirim sebelum output apapun
    header("Access-Control-Allow-Origin: *");
    header("Content-Type: application/json; charset=UTF-8");

    $stmt = $pdo->prepare($query);

    // Bind parameter jika ada pencarian tanggal
    if ($searchDate) {
        $stmt->bindValue(':searchDate', $searchDate);
    }
```

```php
    $stmt->execute();

    $historyData = [];

    while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
        $historyData[] = $row;
    }

    echo json_encode($historyData);
} catch (PDOException $e) {
    echo json_encode(['error' => $e->getMessage()]);
}
?>
```

Gambar 4.70 Kode get_usage_history.php

Gambar 4.70 menunjukkan kode *get_usage_history.php* yang digunakan untuk mengambil data riwayat penggunaan *earplug* dari *database*. File ini merupakan komponen penting dalam sistem karena memberikan data mentah kepada berbagai visualisasi grafik pada tampilan *frontend*, baik grafik mingguan, bulanan, maupun tahunan.

Fungsi utama dari file ini adalah mengeksekusi *query* untuk mengambil seluruh entri pada tabel *history*, yang berisi data penggunaan *earplug* oleh para pengguna. Kode ini diawali dengan menyertakan file *dashboard_connection.php* yang berisi konfigurasi koneksi ke *database* menggunakan PDO (*PHP Data Objects*), sebuah metode yang aman dan fleksibel dalam berinteraksi dengan *database.*

Selanjutnya, kode memeriksa apakah terdapat parameter *searchDate* pada permintaan *GET* dari klien (biasanya dikirim melalui URL). Parameter ini opsional, namun apabila tersedia, sistem akan menambahkan klausa *WHERE* dalam *query SQL* untuk menyaring data berdasarkan tanggal tertentu menggunakan fungsi *DATE()* pada kolom date.

Query SQL yang disusun mencakup pengambilan beberapa kolom penting, seperti *name, no_induk, date, time, rfid_tag, used_quantity, department*, dan *status*.

Format tanggal diformat ulang dalam bentuk YYYY-MM-DD agar seragam dan lebih mudah diolah di sisi klien. Hasil pengambilan data kemudian diurutkan berdasarkan tanggal dan waktu dalam urutan menurun, sehingga data terbaru akan muncul terlebih dahulu.

Proses eksekusi *query* dilakukan dalam blok *try-catch* untuk memastikan jika terjadi kesalahan dalam interaksi *database*, maka pesan kesalahan tetap dapat dikembalikan dalam format *JSON* agar konsisten dengan jenis *response* yang diharapkan klien.

Setelah *query* dieksekusi, setiap baris hasil diambil menggunakan *fetch(PDO::FETCH_ASSOC)* dan disusun ke dalam *array $historyData*. Pada akhir proses, data ini dikonversi ke format *JSON* melalui fungsi *json_encode()* dan dikirim sebagai *response*.

```php
<?php
// File koneksi database
include('dashboard_connection.php');

// Ambil parameter pencarian tanggal jika ada
$searchDate = isset($_GET['searchDate']) ? $_GET['searchDate'] : '';

// Query untuk mengambil data history penggunaan berdasarkan tanggal yang dipilih dan status
$query = "SELECT SUM(used_quantity) AS usedToday
          FROM history
          WHERE DATE(date) = :searchDate AND status = 'normal'";

// Jika tidak ada tanggal yang dipilih, set tanggal default ke hari ini
if (!$searchDate) {
    $searchDate = date('Y-m-d'); // default ke tanggal hari ini
}

try {
    // Siapkan query dan bind tanggal
    $stmt = $pdo->prepare($query);
    $stmt->bindValue(':searchDate', $searchDate);
    $stmt->execute();

    // Ambil hasil
    $result = $stmt->fetch(PDO::FETCH_ASSOC);

    // Kirim data total penggunaan hari ini
    echo json_encode(['usedToday' => $result['usedToday'] ?: 0]);
} catch (PDOException $e) {
    // Menangani error
    echo json_encode(['error' => $e->getMessage()]);
}
?>
```

Gambar 4.71 Kode PHP pada file total_used_today.php

Gambar 4.71 menunjukkan kode PHP pada file *total_used_today.php*, yang bertugas untuk menghitung total jumlah penggunaan *earplug* pada tanggal tertentu. File ini digunakan khusus untuk menyediakan data numerik pada elemen *frontend* yang menampilkan informasi "*Total Used Today*". Kode ini diawali dengan menyertakan file *dashboard_connection.php* sebagai penghubung ke *database* menggunakan pendekatan PDO. Selanjutnya, sistem memeriksa apakah terdapat parameter *searchDate* yang dikirim melalui metode *GET*. Parameter ini bersifat opsional namun apabila tidak tersedia, maka sistem secara otomatis menetapkan tanggal pencarian ke tanggal saat ini menggunakan fungsi date('Y-m-d').

Inti dari proses pengambilan data berada pada *query* SQL yang telah disiapkan. *Query* ini menggunakan fungsi agregat *SUM()* untuk menjumlahkan nilai *used_quantity* dari tabel *history*, dengan syarat bahwa data berada pada tanggal tertentu dan memiliki *status* 'normal'. Kondisi status = 'normal' digunakan untuk memastikan hanya entri yang sah (bukan *limit exceeded* atau kondisi lain yang dianggap tidak valid) yang dihitung.

Query kemudian disiapkan dan dieksekusi menggunakan metode PDO *prepare()* dan *bindValue()* untuk mencegah potensi *SQL injection*. Setelah *query* dieksekusi, hasilnya diambil dan dibaca menggunakan metode *fetch(PDO::FETCH_ASSOC)*. Nilai total penggunaan yang diperoleh kemudian dikemas dalam format *JSON* dan dikirim sebagai *response* ke sisi klien, dengan memastikan bahwa jika hasilnya bernilai null, maka akan dikembalikan sebagai nol (0) melalui operator *ternary* ?:.

Untuk mengantisipasi kemungkinan kesalahan selama proses interaksi dengan database, kode ini juga dilengkapi blok *try-catch*. Apabila terjadi pengecualian (*exception*), maka pesan kesalahan akan dikembalikan ke klien dalam bentuk objek *JSON*, menjaga konsistensi format tanggapan sistem.

```php
<?php
// Include file koneksi database
include('dashboard_connection.php');

// Query untuk mengambil total earplug yang tersedia dari refill
$queryRefill = "SELECT SUM(earplug_count) AS total_refill FROM earplug_refills";
$stmtRefill = $pdo->prepare($queryRefill);
$stmtRefill->execute();
$rowRefill = $stmtRefill->fetch(PDO::FETCH_ASSOC);
$totalRefill = $rowRefill['total_refill'];

// Query untuk menghitung total jumlah scan RFID yang valid
$queryTotalScans = "SELECT COUNT(*) AS total_scans FROM history WHERE status = 'normal'";
$stmtTotalScans = $pdo->prepare($queryTotalScans);
$stmtTotalScans->execute();
$rowTotalScans = $stmtTotalScans->fetch(PDO::FETCH_ASSOC);
$totalScans = $rowTotalScans['total_scans'];  // Setiap scan berarti 1 pasang earplug digunakan

// Total earplug available = total refill - total scans (karena setiap scan berarti 1 pasang digunakan)
$availableEarplug = $totalRefill - $totalScans;

// Kembalikan data dalam format JSON
header("Content-Type: application/json; charset=UTF-8");
echo json_encode(['availableEarplug' => $availableEarplug]);

// Tutup koneksi
$pdo = null;
?>
```

Gambar 4.72 Kode PHP pada file total_earplug.php

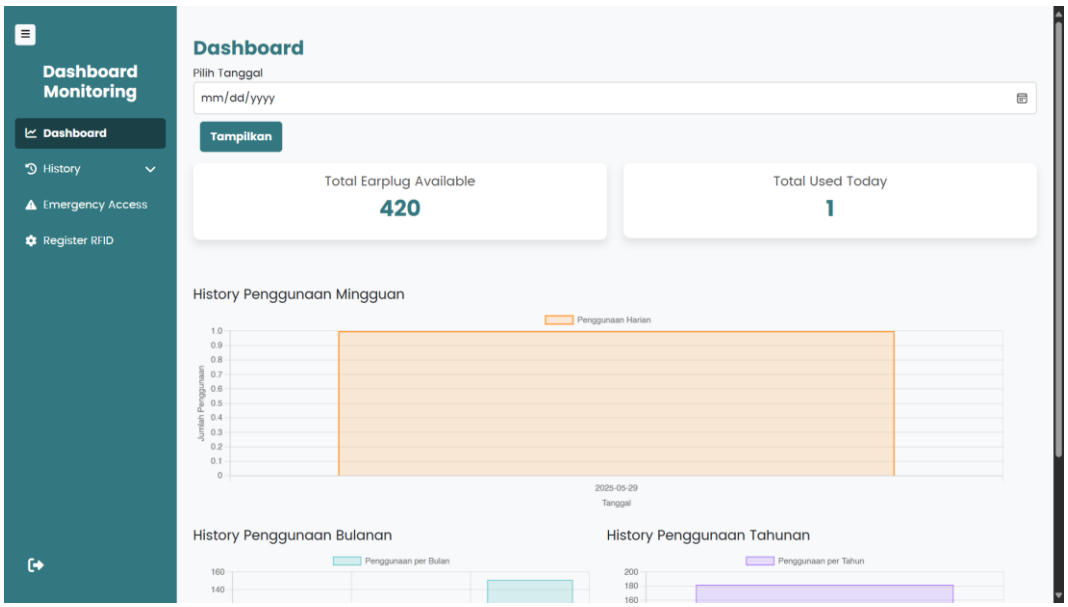
Gambar 4.72 memperlihatkan kode PHP pada *file total_earplug.php*, yang berfungsi untuk menghitung secara dinamis jumlah *earplug* yang masih tersedia di sistem. Nilai ini ditampilkan pada elemen antarmuka dengan label "*Total Earplug Available*", dan diperbarui secara berkala menggunakan fungsi *JavaScript updateTotalEarplugAvailable()*.

Langkah awal dalam kode ini adalah menyertakan *file dashboard_connection.php* sebagai sumber koneksi ke *database*. Selanjutnya, sistem menjalankan dua buah *query* utama dengan tujuan yang berbeda namun saling berkaitan:

1. *Query* pertama (*$queryRefill*) bertugas untuk menghitung total jumlah *earplug* yang telah disuplai atau di-*refill* ke dalam sistem. Data ini diperoleh dari tabel *earplug_refills*, dan dihitung dengan menggunakan fungsi agregat SUM() terhadap kolom *earplug_count*.
2. *Query* kedua *(*$queryTotalScans*)* digunakan untuk menghitung total penggunaan *earplug* yang tercatat secara sah (berstatus 'normal') di dalam tabel

history. Setiap entri dianggap mewakili penggunaan satu pasang *earplug*, karena satu kali pemindaian RFID berarti satu pasang diambil oleh pengguna.

Setelah kedua nilai tersebut berhasil diperoleh, sistem menghitung jumlah *earplug* yang masih tersedia dengan cara mengurangkan *totalRefill* dengan *totalScans*. Hasil pengurangan ini disimpan dalam variabel *$availableEarplug*. Nilai akhir tersebut kemudian dikemas dalam format *JSON* menggunakan fungsi *json_encode(),* dan dikirimkan kembali ke sisi klien. Penetapan *header Content-Type* dilakukan terlebih dahulu untuk memastikan bahwa respon dikenali sebagai data *JSON* oleh browser atau aplikasi antarmuka. Pada Gambar 4.73 ditampilkan Halaman *Dashboard* pada *website*.



Gambar 4.73 Halaman Dashboard Pada Website

4.2.3.3 Pembuatan Program Usage History Page pada Website

Pada tahap ini, dilakukan pembuatan halaman *Usage History* yang berfungsi untuk menampilkan riwayat penggunaan alat pelindung diri, khususnya *earplug*, oleh para pengguna. Bagian ini dirancang menggunakan struktur *HTML* yang disusun secara sistematis untuk memudahkan pengguna dalam melakukan pencatatan dan pemantauan data penggunaan. Elemen utama halaman ini dibungkus dalam suatu *container* dengan atribut *id="usageContent"* dan diberi kelas *content-section*. Elemen tersebut pada awalnya disembunyikan dengan atribut

style="display: none;" agar hanya muncul ketika pengguna memilih menu *Usage History* dari antarmuka aplikasi.

```
<!-- History Usage Content -->
<div id="usageContent" class="content-section" style="display: none;">
    <h3>Usage History</h3>

    <!-- Download Section -->
    <div class="form-group mb-3">
        <!-- Tombol untuk Mengunduh Laporan -->
        <button id="downloadReportButton" class="btn-clear ml-2">Unduh Laporan</button>
    </div>

    <!-- Modal Konfirmasi Unduhan -->
    <div id="confirmationModalOverlay" class="modal-overlay" style="display: none;">
        <div class="modal-content">
            <h5>Apakah anda akan mengunduh data?</h5>
            <div class="form-group">
                <button id="confirmDownload" class="btn btn-primary">Lanjutkan</button>
                <button id="cancelDownload" class="btn btn-secondary ml-2">Batal</button>
            </div>
        </div>
    </div>
```

Gambar 4.74 Kode Button Dan Header Pada Usage History Page

Gambar 4.74 menunjukkan program *button dan header* pada *Usage History Page.* Pada bagian atas halaman, terdapat elemen judul dengan *tag <h3>* yang memberikan informasi bahwa pengguna sedang berada pada halaman riwayat penggunaan. Selanjutnya, disediakan sebuah tombol dengan label "Unduh Laporan" yang memiliki *atribut id="downloadReportButton"*. Tombol ini berfungsi untuk memulai proses pengunduhan laporan riwayat penggunaan dalam format tertentu, dan secara visual didesain dengan kelas *btn-clear* agar selaras dengan gaya keseluruhan antarmuka.

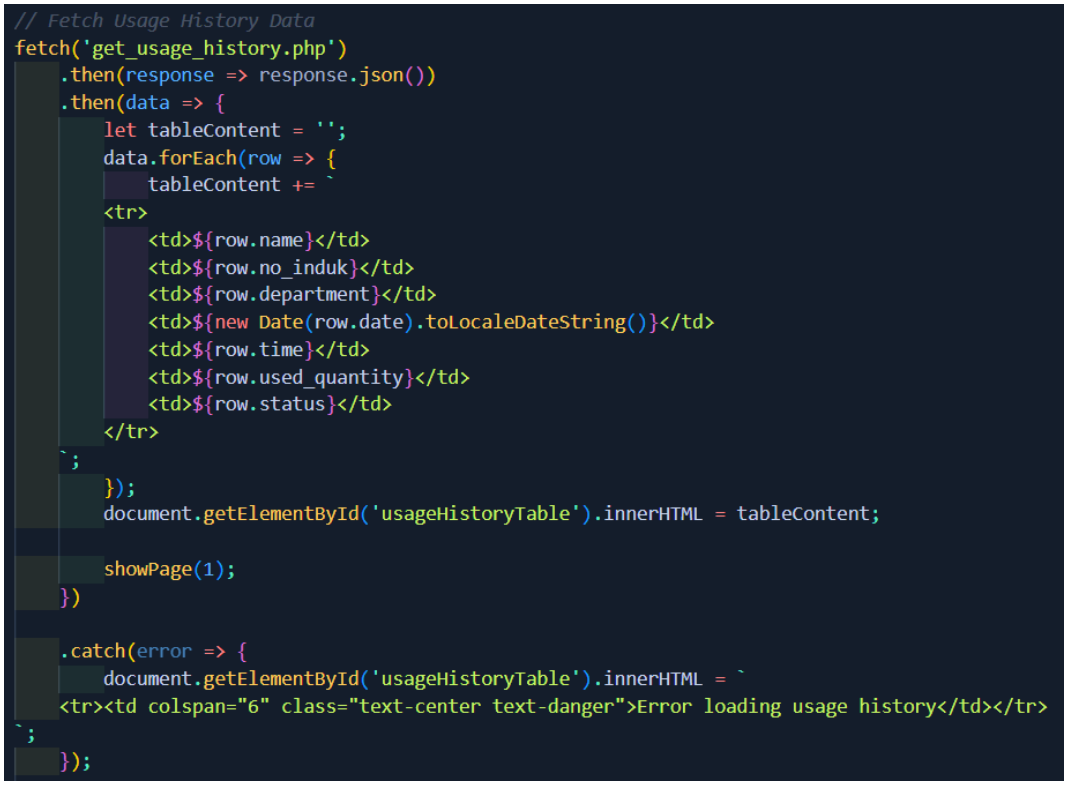
Untuk memberikan pengalaman pengguna yang lebih aman, sistem juga menyediakan sebuah jendela konfirmasi (*modal*) yang muncul ketika pengguna mengklik tombol unduh. Jendela ini diberi *id="confirmationModalOverlay"* dan disembunyikan secara default. Di dalamnya terdapat pertanyaan konfirmasi serta dua tombol tindakan, yaitu "Lanjutkan" untuk melanjutkan proses unduhan dan "Batal" untuk membatalkan tindakan tersebut. Fitur ini bertujuan untuk mencegah tindakan unduhan yang tidak disengaja oleh pengguna.

```
<!-- Table Section -->
<div id="usageHistoryTableWrapper">
    <table class="table table-striped">
        <thead>
            <tr>
                <th onclick="sortUsageTable(0)">Name <i class="fas fa-sort"></i></th>
                <th onclick="sortUsageTable(1)">No. Induk <i class="fas fa-sort"></i></th>
                <th onclick="sortUsageTable(2)">Department <i class="fas fa-sort"></i></th>
                <th onclick="sortUsageTable(3)">Date <i class="fas fa-sort"></i></th>
                <th onclick="sortUsageTable(4)">Time <i class="fas fa-sort"></i></th>
                <th onclick="sortUsageTable(5)">Used Quantity <i class="fas fa-sort"></i></th>
                <th onclick="sortUsageTable(6)">Status <i class="fas fa-sort"></i></th>
            </tr>
        </thead>
        <tbody id="usageHistoryTable">
            <!-- Data Usage Akan Diisi Secara Dinamis -->
        </tbody>
    </table>
    <div id="paginationControls" class="mt-3">
        <!-- Tombol pagination akan di-generate lewat JS -->
    </div>
    <div id="paginationInfo" style="text-align:center; margin-top: 0.5rem;"></div>
</div>
</div>
```

Gambar 4.75 Tabel pada Usage History Page

Selanjutnya, pada Gambar 4.75 ditampilkan sebuah tabel dengan kelas *table table-striped* yang digunakan untuk menyajikan data riwayat penggunaan secara terstruktur. Tabel ini memiliki tujuh kolom utama, yaitu *Name*, *No. Induk*, *Department*, *Date*, *Time*, *Used Quantity*, dan *Status*. Masing-masing kolom dilengkapi dengan ikon sortir dari pustaka *Font Awesome* serta fungsi *onclick* bernama *sortUsageTable(index)* yang memungkinkan pengguna mengurutkan data berdasarkan kolom yang diinginkan. Seluruh data pada tabel ini tidak dituliskan secara statis, melainkan akan diisi secara dinamis oleh *JavaScript* ke dalam elemen *<tbody>* dengan *id="usageHistoryTable"*.

Sebagai pelengkap tampilan tabel, disediakan pula dua elemen tambahan, yaitu *paginationControls* dan *paginationInfo*. Kedua elemen ini dirancang untuk memfasilitasi pembagian data menjadi beberapa halaman apabila jumlah data yang ditampilkan terlalu banyak, sehingga tetap memberikan kenyamanan dalam navigasi. Elemen-elemen ini akan diisi secara dinamis melalui fungsi yang ditulis dalam skrip *JavaScript*.



```
// Fetch Usage History Data
fetch('get_usage_history.php')
    .then(response => response.json())
    .then(data => {
        let tableContent = '';
        data.forEach(row => {
            tableContent += `
        <tr>
            <td>${row.name}</td>
            <td>${row.no_induk}</td>
            <td>${row.department}</td>
            <td>${new Date(row.date).toLocaleDateString()}</td>
            <td>${row.time}</td>
            <td>${row.used_quantity}</td>
            <td>${row.status}</td>
        </tr>
    `;
        });
        document.getElementById('usageHistoryTable').innerHTML = tableContent;

        showPage(1);
    })

    .catch(error => {
        document.getElementById('usageHistoryTable').innerHTML = `
    <tr><td colspan="6" class="text-center text-danger">Error loading usage history</td></tr>
`;
    });
```

Gambar 4.76 Kode yang Mengatur Pengambilan Data Riwayat Penggunaan

Pengolahan data riwayat penggunaan (*usage history*) pada sistem ini dilakukan menggunakan bahasa pemrograman *JavaScript* yang memanfaatkan metode *fetch* untuk mengambil data dari server secara *asynchronous*. Data yang diambil berasal dari berkas get_usage_history.php dan diharapkan dalam format *JavaScript Object Notation* (*JSON*). Gambar 4.76 menunjukkan potongan kode JavaScript yang mengatur proses pengambilan data riwayat penggunaan, sedangkan Gambar X merupakan kode penerapan logika penyaringan berdasarkan nama. Setelah berhasil diperoleh, data tersebut diproses dan ditampilkan ke dalam elemen tabel HTML melalui pemanggilan *document.getElementById('usageHistoryTable').innerHTML*. Setiap entri data ditampilkan dalam bentuk baris tabel yang mencakup atribut nama pengguna, nomor induk, departemen, tanggal, waktu, jumlah *earplug* yang digunakan, serta status penggunaan. Untuk meningkatkan keterbacaan, data tanggal diformat menggunakan metode bawaan *JavaScript* yaitu *toLocaleDateString()* yang

menyesuaikan tampilan tanggal dengan format lokal pengguna. Selain itu, apabila terjadi kesalahan dalam proses pengambilan data misalnya akibat kegagalan koneksi atau respons *server* yang tidak valid, sistem akan menampilkan pesan kesalahan langsung pada tampilan tabel, sehingga pengguna dapat segera mengetahui adanya gangguan sistem.

```
//pagination halaman usage history
  const rowsPerPage = 10; // jumlah data per halaman
  let currentPage = 1;

  function showPage(page) {
    const rows = document.querySelectorAll('#usageHistoryTable tr');
    const totalRows = rows.length;
    const totalPages = Math.ceil(totalRows / rowsPerPage);

    // Validasi halaman
    if (page < 1) page = 1;
    if (page > totalPages) page = totalPages;

    currentPage = page;

    // Menyembunyikan/menampilkan baris sesuai dengan halaman
    rows.forEach((row, index) => {
        row.style.display = (index >= (page - 1) * rowsPerPage && index < page * rowsPerPage) ? '' : 'none';
    });

    // Memperbarui kontrol pagination
    renderPaginationControls(totalPages);

    // Menampilkan informasi jumlah data yang sedang ditampilkan
    const start = (page - 1) * rowsPerPage + 1;
    const end = Math.min(page * rowsPerPage, totalRows);
    document.getElementById('paginationInfo').innerHTML = `Showing ${start} to ${end} of ${totalRows} entries`;
}
```

```
function renderPaginationControls(totalPages) {
    const container = document.getElementById('paginationControls');
    container.innerHTML = ''; // Clear previous controls

    if (totalPages <= 1) return;

    const prevButton = document.createElement('button');
    prevButton.innerHTML = '<i class="fas fa-chevron-left"></i>'; // Menambahkan ikon panah kiri
    prevButton.disabled = currentPage === 1;
    prevButton.onclick = () => showPage(currentPage - 1);
    container.appendChild(prevButton);

    // Tombol halaman
    for (let i = 1; i <= totalPages; i++) {
        const pageButton = document.createElement('button');
        pageButton.textContent = i;
        if (i === currentPage) pageButton.disabled = true;
        pageButton.onclick = () => showPage(i);
        container.appendChild(pageButton);
    }

    const nextButton = document.createElement('button');
    nextButton.innerHTML = '<i class="fas fa-chevron-right"></i>'; // Menambahkan ikon panah kanan
    nextButton.disabled = currentPage === totalPages;
    nextButton.onclick = () => showPage(currentPage + 1);
    container.appendChild(nextButton);
}
```

Gambar 4.77 Kode Fitur Pagination Pada Halaman Usage History

Agar data riwayat penggunaan (*usage history*) dapat ditampilkan dengan lebih terstruktur dan mudah diakses oleh pengguna, sistem ini menerapkan fitur *pagination* menggunakan bahasa pemrograman *JavaScript* seperti pada kode yang tertera di Gambar 4.77. Pada kode ini, jumlah baris yang ditampilkan per halaman ditetapkan sebesar 10 baris melalui variabel *rowsPerPage*. Fungsi utama yang digunakan untuk mengatur tampilan halaman adalah *showPage(page),* yang akan menampilkan baris-baris data sesuai dengan halaman yang dipilih oleh pengguna. Fungsi ini akan terlebih dahulu menghitung total jumlah baris yang tersedia dan jumlah halaman yang dibutuhkan menggunakan *Math.ceil(totalRows / rowsPerPage).* Selanjutnya, fungsi tersebut akan menyembunyikan semua baris yang tidak termasuk dalam rentang indeks halaman yang aktif, dan hanya menampilkan baris yang sesuai. Selain itu, fungsi *showPage()* juga akan memperbarui informasi jumlah data yang sedang ditampilkan di bagian bawah tabel, misalnya dalam format "*Showing 11 to 20 of 50 entries*", yang diatur melalui elemen dengan *ID paginationInfo*.

Untuk mengatur navigasi antar halaman, fungsi *renderPaginationControls(totalPages)* digunakan. Fungsi ini akan menghasilkan elemen tombol navigasi yang mencakup tombol panah untuk ke halaman sebelumnya dan berikutnya, serta tombol-tombol bernomor untuk tiap halaman yang tersedia. Masing-masing tombol dibuat secara dinamis menggunakan *document.createElement('button'),* dan setiap tombol diberikan aksi ketika diklik untuk memanggil fungsi *showPage(i)* dengan parameter halaman yang sesuai. Dengan adanya implementasi ini, pengguna dapat menelusuri data dalam jumlah besar dengan lebih efisien tanpa harus melihat seluruh data dalam satu tampilan yang memanjang.

```
// Tombol "Unduh Laporan" membuka modal input rentang tanggal
document.addEventListener("DOMContentLoaded", function () {
    // Menambahkan event listener untuk tombol "Unduh Laporan"
    document.getElementById("downloadReportButton").addEventListener("click", function () {
        // Menampilkan modal konfirmasi
        document.getElementById("confirmationModalOverlay").style.display = "flex";
    });

    // Menambahkan event listener untuk tombol "Lanjutkan" (konfirmasi)
    document.getElementById("confirmDownload").addEventListener("click", function () {
        // Menutup modal konfirmasi
        document.getElementById("confirmationModalOverlay").style.display = "none";

        // Lakukan pengunduhan laporan
        var xhr = new XMLHttpRequest();
        xhr.open("GET", "export_data.php", true); // Tidak perlu rentang tanggal lagi
        xhr.responseType = 'blob';

        xhr.onload = function () {
            if (xhr.status === 200) {
                var blob = xhr.response;
                var link = document.createElement('a');
                link.href = URL.createObjectURL(blob);
                link.download = "laporan_earplug_full_data.xlsx"; // Nama file yang diunduh
                document.body.appendChild(link);
                link.click();
                document.body.removeChild(link);
                URL.revokeObjectURL(link.href);
            } else {
                console.error("Gagal mengunduh laporan: " + xhr.status);
                alert("Terjadi kesalahan saat mengunduh laporan.");
            }
        };
```

Gambar 4.78 Kode Unduh Laporan Pada Halaman Usage History

Untuk mendukung kebutuhan dokumentasi dan pelaporan data penggunaan alat pelindung telinga, sistem ini dilengkapi dengan fitur Unduh Laporan yang memungkinkan pengguna untuk mengunduh seluruh data *usage history* dalam format berkas Excel. Pada Gambar 4.78 ditunjukkan kode untuk membangun fitur unduh laporan. Fitur ini diinisialisasi melalui penekanan tombol dengan label "Unduh Laporan", yang selanjutnya akan memunculkan *modal overlay* berupa jendela konfirmasi. Interaksi ini dikendalikan oleh penangan peristiwa (*event listener*) yang diaktifkan ketika dokumen selesai dimuat (*DOMContentLoaded*), dengan mengaitkan fungsi tertentu pada elemen dengan *identifier downloadReportButton*.

Apabila pengguna mengonfirmasi dengan menekan tombol "Lanjutkan", maka sistem akan mengirimkan permintaan *asinkron* ke peladen menggunakan

objek *XMLHttpRequest*. Permintaan ini ditujukan ke berkas export_data.php, dan dikonfigurasi dengan jenis tanggapan (*response type*) berupa *blob*, yang menandakan bahwa peladen akan mengembalikan data biner dalam bentuk berkas Excel. Setelah tanggapan berhasil diterima dengan status 200, skrip akan secara dinamis membuat elemen *hyperlink* (<a>) untuk memicu pengunduhan berkas dengan nama *laporan_earplug_full_data.xlsx*. Apabila terjadi kesalahan, baik pada saat permintaan maupun tanggapan, pengguna akan menerima pesan kesalahan melalui *alert* atau pencatatan kesalahan di konsol (*console log*).

Sebaliknya, apabila pengguna memilih untuk membatalkan proses dengan menekan tombol “Batal”, maka *modal overlay* akan ditutup tanpa melanjutkan proses permintaan data.

Kode pada lampiran 15 merupakan skrip *server-side* berbasis PHP yang digunakan untuk menghasilkan laporan data penggunaan *earplug* dalam format *spreadsheet* Excel. Pada bagian awal, skrip mengatur zona waktu ke *Asia/Jakarta* serta memuat pustaka eksternal *PhpSpreadsheet* melalui *autoload* dari *Composer*. Beberapa kelas yang di-*import* dari pustaka tersebut antara lain adalah Spreadsheet, Xlsx, Fill, *Border*, dan *Alignment*, yang digunakan untuk membentuk struktur, gaya, serta *output* dokumen Excel.

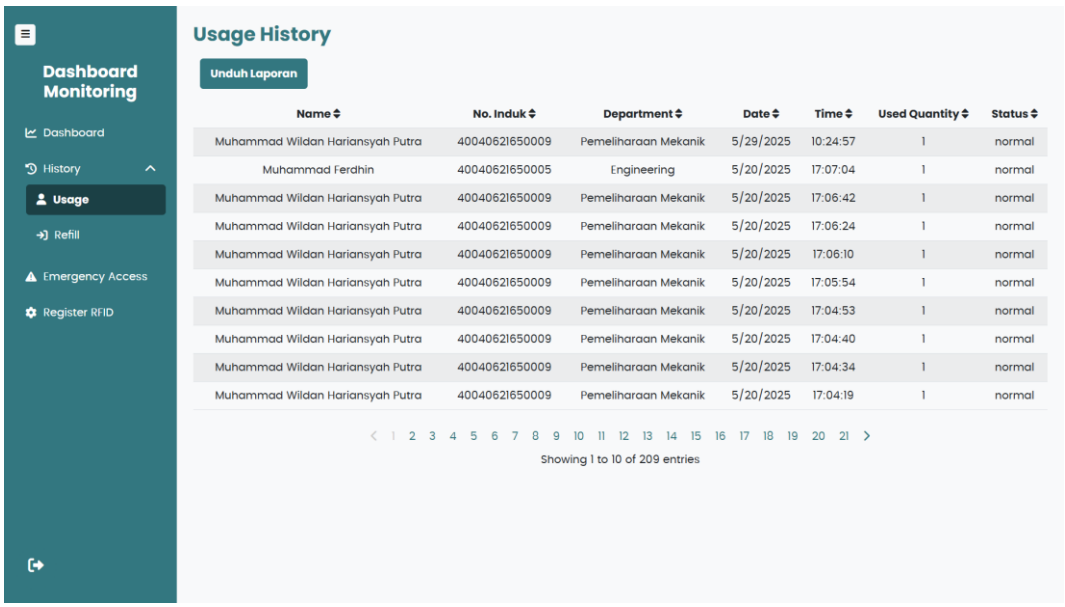
Selanjutnya, dilakukan validasi terhadap masukan tanggal melalui parameter *GET* bernama *from* dan *to*, untuk memastikan format tanggal sesuai pola *YYYY-MM-DD*. Apabila parameter tidak tersedia atau tidak valid, maka digunakan nilai bawaan mulai dari 1 Januari 2025 hingga tanggal saat ini. Setelah itu, sistem melakukan koneksi ke basis data *MySQL* menggunakan ekstensi *mysqli*, lalu menjalankan tiga perintah *query* untuk mengambil data dari tabel *history*, *earplug_refills*, dan *rfid_users* yang aktif.

Data yang telah diambil kemudian digunakan untuk menghitung total penggunaan *earplug* dan total *refill* yang dilakukan, dengan mengecualikan entri yang berstatus *limit_exceeded*. Proses selanjutnya adalah membentuk empat lembar kerja (*sheet*) dalam objek Spreadsheet. *Sheet* pertama berjudul "*History*" menyajikan riwayat penggunaan berdasarkan nama pengguna, nomor induk, departemen, waktu pemakaian, serta status. Baris dengan status *limit_exceeded*

diberi warna latar merah muda sebagai penanda visual. *Sheet* kedua "Refill" menampilkan data pengisian ulang, seperti jumlah isi ulang, ID admin, tanggal, dan sisa alat pelindung. *Sheet* ketiga "RFID Users" berisi daftar pengguna aktif lengkap dengan ID, nama, divisi, batas tap, dan waktu pembuatan akun.

Setiap lembar kerja diformat secara otomatis agar lebar kolom menyesuaikan konten dan diberikan gaya tabel menggunakan fungsi *applyTableStyle()* yang mengatur batas (*border*), perataan (*alignment*), dan pembungkusan teks (*wrap text*). *Sheet* keempat berjudul "Ringkasan" menyajikan informasi singkat seperti waktu ekspor, periode data, total pemakaian dan *refill*, serta jumlah pengguna aktif. Lembar ini ditetapkan sebagai lembar aktif agar ditampilkan pertama kali saat berkas dibuka.

Sebagai penutup, skrip mengatur jenis konten (*content-type*) menjadi format Excel dan mengirimkan file hasil ekspor secara langsung kepada pengguna dengan nama berkas laporan_earplug_[*from*]_sd_[*to*].xlsx, di mana [*from*] dan [*to*] merepresentasikan rentang tanggal yang dipilih. Dengan demikian, skrip ini mendukung dokumentasi kegiatan pemakaian alat pelindung secara sistematis, rapi, dan dapat diarsipkan dalam format yang kompatibel dengan perangkat lunak *spreadsheet* seperti Microsoft Excel atau *Google Sheets*. Pada Gambar 4.79 ditunjukkan Halaman *Usage History*.



Gambar 4.79 Halaman Usage History Pada Website

4.2.3.4 Pembuatan Program Refill History Page pada Website

Pada tahap ini dilakukan pembuatan halaman *Refill History* yang memiliki fungsi untuk melakukan *input* jumlah *refill* dan menampilkan rekap data *input* dalam bentuk tabel dan grafik.

```
<!-- Refill Content (Refill History dan Input Refill Form) -->
<div id="refillContent" class="content-section" style="display: none;">

    <!-- Form Input Refill -->
    <form id="refillForm" method="POST" action="submit_refill.php">
        <div class="row">
            <div class="col-md-6">
                <div class="mb-3">
                    <label for="refillDate" class="form-label">Tanggal Refill</label>
                    <input type="date" class="form-control" id="refillDate" name="refillDate" required>
                </div>
            </div>
            <div class="col-md-6">
                <div class="mb-3">
                    <label for="refillQuantity" class="form-label">Jumlah Refill</label>
                    <input type="number" class="form-control" id="refillQuantity" name="refillQuantity"
                        required min="1" max="350">
                </div>
            </div>
        </div>
        <button type="submit" class="btn-submit">Submit</button>
        <button type="button" id="clearFormBtn" class="btn-clear ml-2">Clear</button>
    </form>
```

Gambar 4.80 Kode Antarmuka Halaman Refill History

Pada Gambar 4.80 ditampilkan program antarmuka halaman *Refill History*, yang mencakup dua bagian utama, yaitu *form input refill* dan *tabel riwayat refill* beserta visualisasi datanya. Elemen ini dibungkus dalam sebuah *container* dengan *id="refillContent"* dan diberi kelas *content-section*. Sama seperti halaman *Usage History*, bagian ini disembunyikan secara *default* melalui atribut *style="display: none;"* dan hanya akan ditampilkan apabila pengguna memilih menu yang sesuai dari antarmuka.

Bagian pertama dari konten ini adalah *form input refill* yang memungkinkan pengguna mencatat penambahan jumlah *earplug* yang dimasukkan ke dalam dispenser. Formulir ini disusun menggunakan *tag* <*form*> dengan metode *POST* dan diarahkan ke berkas *submit_refill.php*. Formulir tersebut terdiri atas dua buah isian, yaitu *Tanggal Refill* dan *Jumlah Refill*. Untuk *Tanggal Refill*, digunakan elemen <*input*> dengan tipe *date* yang memudahkan pengguna dalam memilih tanggal melalui antarmuka kalender. Sementara itu, *Jumlah Refill* menggunakan

tipe *number* dan dilengkapi dengan atribut *min="1"* dan *max="350"* untuk membatasi *input* agar tetap dalam rentang yang diizinkan.

Tersedia pula dua tombol aksi, yaitu tombol *Submit* untuk mengirim *data refill* yang dimasukkan, serta tombol *Clear* dengan *id="clearFormBtn"* yang berfungsi untuk menghapus atau mereset isian pada formulir. Tombol ini secara visual dibedakan dengan kelas *btn-clear*, agar tampilannya sesuai dengan gaya tombol lainnya di dalam sistem.

Selanjutnya, bagian *Refill History* disusun untuk menampilkan data histori pengisian ulang yang telah dilakukan. Data ditampilkan dalam bentuk tabel menggunakan *tag* <*table*> dengan kelas *table table-striped*. Tabel ini memiliki tiga kolom utama, yaitu *Date*, *Earplug Count*, dan *Action*. Kolom *Action* nantinya akan memuat tombol interaktif seperti hapus atau edit yang diatur menggunakan *JavaScript*. Seluruh data ditampilkan secara dinamis ke dalam elemen <*tbody*> dengan *id="refillHistoryTable"*. Gambar 4.81 merupakan kode untuk membangun tabel pada *Refill History*.

```
<!-- Refill History Table -->
<div class="d-flex flex-column" id="refillHistoryWrapper">

    <div id="refillHistoryTableWrapper">
        <table class="table table-striped">
            <thead>
                <tr>
                    <th>Date</th>
                    <th>Earplug Count</th>
                    <th>Action</th>
                </tr>
            </thead>
            <tbody id="refillHistoryTable">
                <!-- Data History Refill Akan Ditampilkan Di Sini -->
            </tbody>
        </table>
    </div>
```

Gambar 4.81 Kode Tabel pada Halaman Refill History

Sebagai pelengkap fitur monitoring, disediakan pula dua buah grafik seperti pada Gambar 4.82 yang memberikan visualisasi data refill, yaitu grafik bulanan (*monthlyChart*) dan grafik tahunan (*annualChart*). Keduanya menggunakan *tag*

<*canvas*> dan akan digambar secara dinamis melalui pustaka *JavaScript* seperti *Chart.js* untuk menampilkan tren pengisian ulang *earplug* dari waktu ke waktu. Grafik ini ditujukan untuk membantu pengguna atau pengelola sistem dalam melakukan analisis terhadap frekuensi dan volume pengisian ulang yang terjadi.

```
<!-- Refill History Chart -->
<div class="row mt-4">
    <!-- Chart Bulanan -->
    <div class="col-md-6">
        <h5>Refill Bulanan</h5>
        <canvas id="monthlyChart" width="400" height="200"></canvas>
    </div>

    <!-- Chart Tahunan -->
    <div class="col-md-6">
        <h5>Refill Tahunan</h5>
        <canvas id="annualChart" width="400" height="200"></canvas>
    </div>
</div>
</div>
</div>
```

Gambar 4.82 Kode Untuk Menampilkan Chart Pada Halaman Refill History

Melalui fitur ini, pengguna tidak hanya dapat mencatat pengisian ulang, tetapi juga melakukan pemantauan historis secara lebih interaktif dan informatif. Penggunaan tabel dan grafik secara bersamaan memperkuat aspek pelaporan dan pengambilan keputusan dalam pengelolaan logistik *earplug* di lingkungan kerja. Untuk mendukung proses visualisasi data serta menampilkan informasi *refill* secara dinamis pada antarmuka, diterapkan suatu mekanisme pemrograman menggunakan JavaScript yang berfungsi mengambil dan mengolah data dari sisi *backend* untuk kemudian ditampilkan dalam bentuk tabel dan grafik secara real-time.

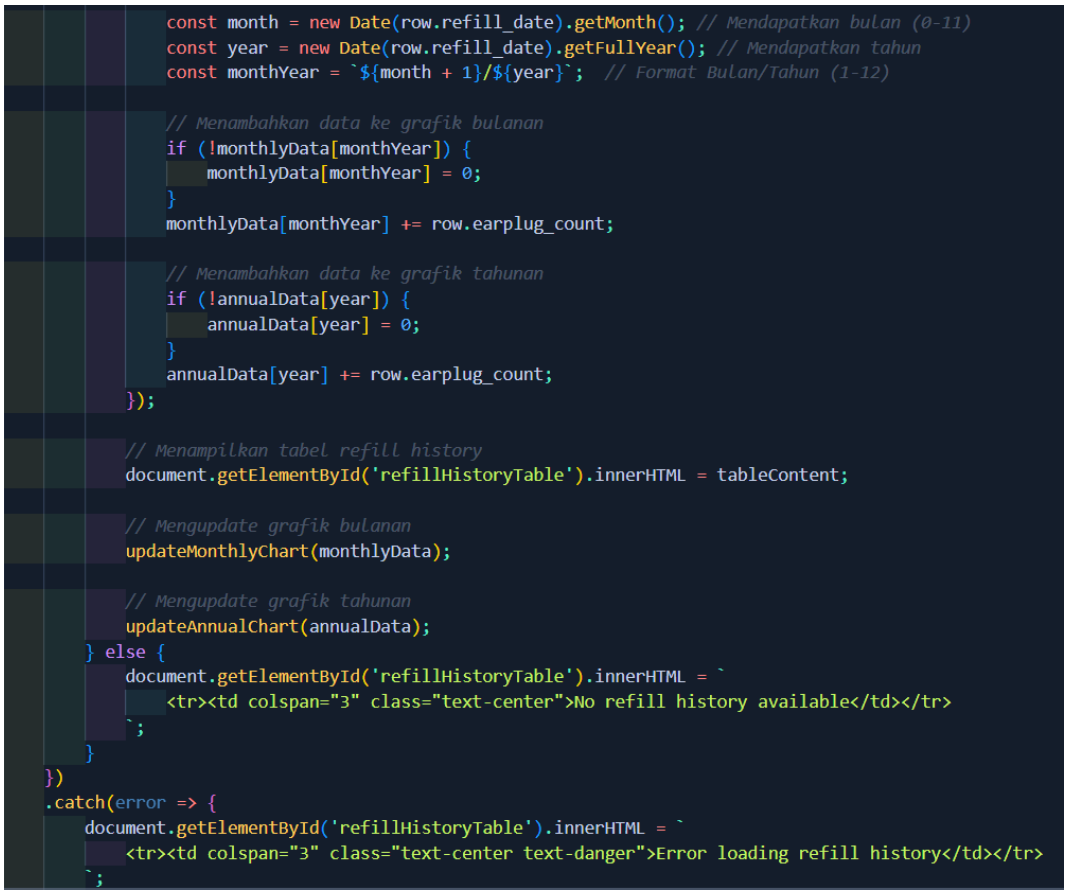
```
// Fetch refill history data
fetch('get_refill_history.php')
    .then(response => response.json())  // Parse as JSON
    .then(data => {
        refillData = data; // Menyimpan data refill secara global
        let tableContent = '';
        let monthlyData = {};
        let annualData = {};

        // Mengurutkan data berdasarkan tanggal refill
        data.sort((a, b) => new Date(a.refill_date) - new Date(b.refill_date));

        if (data.length > 0) {
            data.forEach((row, index) => {
                const dateObj = new Date(row.refill_date);
                const formattedDate = `${String(dateObj.getDate()).padStart(2, '0')}-${String(dateObj.getMonth() + 1).
                                       padStart(2, '0')}-${dateObj.getFullYear()}`;

                tableContent += `
                    <tr id="refill-row-${row.refill_id}">
                        <td>${formattedDate}</td>
                        <td>${row.earplug_count}</td>
                        <td>
                            <button class="btn btn-danger" onclick="deleteRefill(${row.refill_id})">Delete</button>
                        </td>
                    </tr>
                `;
```

Gambar 4.83 Kode Fetch untuk Data Refill History

Gambar 4.83 merupakan kode *fetch* untuk data *refill history*. Kode ini diawali dengan deklarasi *array* kosong bernama *refillData* yang berfungsi untuk menyimpan data *refill* secara *global*, sehingga dapat digunakan untuk referensi saat melakukan penghapusan data. Selanjutnya, dilakukan proses pengambilan data melalui metode *fetch()* dari *file get_refill_history.php*, yang kemudian diolah dalam format JSON menggunakan *.then(response => response.json()*). Apabila proses pengambilan berhasil, data yang diterima akan disimpan ke dalam variabel *refillData*, dan sejumlah variabel tambahan seperti *tableContent*, *monthlyData*, dan *annualData* diinisialisasi untuk menyusun konten tabel dan grafik.

Data yang diperoleh kemudian diurutkan berdasarkan tanggal *refill* secara menaik menggunakan fungsi *sort()*. Setelah itu, dilakukan iterasi terhadap setiap entri data menggunakan metode *forEach(),* di mana setiap baris data akan dikonversi tanggalnya ke dalam format DD-MM-YYYY agar lebih mudah dibaca oleh pengguna. Masing-masing entri disusun menjadi elemen <tr> dalam bentuk HTML dan dimasukkan ke dalam variabel *tableContent*. Selain itu, pada setiap iterasi juga dilakukan proses agregasi jumlah earplug berdasarkan bulan dan tahun, yang kemudian disimpan dalam objek *monthlyData* dan *annualData*. Proses ini bertujuan untuk menghasilkan dua jenis visualisasi grafik, yaitu grafik bulanan dan

tahunan, berdasarkan tren pengisian ulang earplug. Setelah seluruh data diproses, isi dari *tableContent* ditampilkan ke dalam elemen *<tbody>* dengan id *refillHistoryTable*, sehingga pengguna dapat melihat riwayat pengisian ulang secara langsung dalam bentuk tabel.



```
            const month = new Date(row.refill_date).getMonth(); // Mendapatkan bulan (0-11)
            const year = new Date(row.refill_date).getFullYear(); // Mendapatkan tahun
            const monthYear = `${month + 1}/${year}`;  // Format Bulan/Tahun (1-12)

            // Menambahkan data ke grafik bulanan
            if (!monthlyData[monthYear]) {
                monthlyData[monthYear] = 0;
            }
            monthlyData[monthYear] += row.earplug_count;

            // Menambahkan data ke grafik tahunan
            if (!annualData[year]) {
                annualData[year] = 0;
            }
            annualData[year] += row.earplug_count;
        });

        // Menampilkan tabel refill history
        document.getElementById('refillHistoryTable').innerHTML = tableContent;

        // Mengupdate grafik bulanan
        updateMonthlyChart(monthlyData);

        // Mengupdate grafik tahunan
        updateAnnualChart(annualData);
    } else {
        document.getElementById('refillHistoryTable').innerHTML = `
            <tr><td colspan="3" class="text-center">No refill history available</td></tr>
        `;
    }
})
.catch(error => {
    document.getElementById('refillHistoryTable').innerHTML = `
        <tr><td colspan="3" class="text-center text-danger">Error loading refill history</td></tr>
    `;
```

Gambar 4.84 Fungsi untuk Update Grafik Pada Halaman Refill History

Pada Gambar 4.84 ditunjukkan kode untuk *update* grafik bulanan dan tahunan pada halaman *Refill History.* Terdapat dua fungsi bernama *updateMonthlyChart()* dan *updateAnnualChart()* yang dipanggil untuk memperbarui grafik visualisasi berdasarkan data agregat yang telah disusun sebelumnya. Apabila data yang diambil kosong, maka akan ditampilkan baris dengan pesan bahwa riwayat pengisian ulang tidak tersedia. Sementara itu, jika terjadi kesalahan selama proses *fetching*, maka ditampilkan pesan kesalahan dalam elemen tabel yang sama. Dengan demikian, kode ini tidak hanya menampilkan data secara informatif, tetapi

juga memastikan bahwa pengguna tetap mendapatkan umpan balik meskipun terjadi kondisi gagal muat data.

```
// Fungsi untuk memperbarui grafik bulanan
function updateMonthlyChart(monthlyData) {
    const months = Object.keys(monthlyData);
    const counts = months.map(month => monthlyData[month]);

    const ctx = document.getElementById('monthlyChart').getContext('2d');
    new Chart(ctx, {
        type: 'line',
        data: {
            labels: months,
            datasets: [{
                label: 'Earplug Count per Month',
                data: counts,
                borderColor: 'rgb(75, 192, 192)',
                fill: false,
            }]
        },
        options: {
            responsive: true,
            scales: {
                x: {
                    title: {
                        display: true,
                        text: 'Month/Year'
                    }
                },
                y: {
                    title: {
                        display: true,
                        text: 'Earplug Count'
                    },
                    beginAtZero: true
                }
            }
        }
    });
}
```

Gambar 4.85 Kode updateMonthlyChart()

Gambar 4.85 menunjukkan kode fungsi *updateMonthlyChart()* yang bertujuan untuk menampilkan tren pengisian ulang *earplug* dalam satuan waktu bulanan menggunakan grafik garis berbasis Chart.js. Parameter *monthlyData* yang diterima oleh fungsi ini adalah sebuah objek *JavaScript* yang berisi pasangan *key-value*, di mana kunci merepresentasikan bulan dan tahun (misalnya "01-2024") dan nilai merupakan jumlah earplug yang di-*refill* pada bulan tersebut.

Langkah pertama dalam fungsi ini adalah mengambil seluruh kunci dari objek *monthlyData* menggunakan *Object.keys(monthlyData),* dan menyimpannya ke

dalam variabel *months*. Nilai-nilai dari masing-masing kunci tersebut diambil dan disimpan dalam variabel *counts* menggunakan fungsi *map()*. Dengan cara ini, *months* akan berisi *array label* sumbu-X berupa waktu, sedangkan *counts* akan berisi *array* nilai sumbu-Y berupa jumlah *earplug*.

Selanjutnya, konteks dari elemen <*canvas*> dengan *id monthlyChart* diambil menggunakan *getContext('2d')* untuk membuat area gambar dua dimensi yang dibutuhkan oleh Chart.js. Kemudian, grafik baru dibuat menggunakan konstruktor *Chart()*, dengan tipe *line* yang berarti grafik garis.

```
// Fungsi untuk memperbarui grafik tahunan
function updateAnnualChart(annualData) {
    const years = Object.keys(annualData);
    const counts = years.map(year => annualData[year]);

    const ctx = document.getElementById('annualChart').getContext('2d');
    new Chart(ctx, {
        type: 'line',
        data: {
            labels: years,
            datasets: [{
                label: 'Earplug Count per Year',
                data: counts,
                borderColor: '#eb9834',
                fill: false,
            }]
        },
        options: {
            responsive: true,
            scales: {
                x: {
                    title: {
                        display: true,
                        text: 'Year'
                    }
                },
                y: {
                    title: {
                        display: true,
                        text: 'Earplug Count'
                    },
                    beginAtZero: true
                }
            }
        }
    });
```

Gambar 4.86 Kode updateAnnualChart()

Fungsi updateAnnualChart() pada Gambar 4.86 digunakan untuk menampilkan data jumlah *refill earplug* dalam skala tahunan melalui grafik garis yang interaktif. Fungsi ini bekerja dengan menerima parameter berupa objek

annualData, yang berisi pasangan *key-value* di mana *key* merupakan tahun (seperti "2023", "2024") dan *value* adalah jumlah *earplug* yang tercatat pada tahun tersebut. Data tahun kemudian diambil menggunakan *Object.keys()* dan disimpan dalam *variabel years*, sementara jumlah *earplug* untuk masing-masing tahun disimpan dalam *variabel counts* menggunakan metode *.map()*.

Selanjutnya, fungsi ini mengakses elemen *canvas* HTML dengan id *annualChart* dan mendapatkan konteks 2D-nya sebagai tempat menggambar grafik menggunakan pustaka *Chart.js*. Grafik yang ditampilkan bertipe *line* atau garis, dengan sumbu X berisi label tahun dan sumbu Y menunjukkan jumlah earplug. Warna garis grafik diatur menggunakan kode warna oranye #eb9834, dan area di bawah garis tidak diisi warna (*fill: false*).

Dalam bagian opsi grafik (*options*), pengaturan *responsive* diaktifkan agar grafik dapat menyesuaikan ukuran layar pengguna. Selain itu, sumbu X dan Y masing-masing diberi judul "*Year*" dan "*Earplug Count*", serta sumbu Y dimulai dari angka nol (*beginAtZero: true*) untuk memberikan skala yang akurat.

```
// Fungsi untuk menghapus data refill
function deleteRefill(refillId) {
    // Menanyakan konfirmasi pengguna sebelum menghapus data
    const confirmDelete = confirm("Are you sure you want to delete this refill record?");
    if (!confirmDelete) {
        return;  // Jika pengguna membatalkan, keluar dari fungsi
    }

    // Menghapus data refill berdasarkan refill_id
    fetch('delete_refill.php', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({ refill_id: refillId })
    })
        .then(response => response.json())
        .then(result => {
            if (result.success) {
                // Refresh halaman untuk mendapatkan data terbaru
                location.reload();  // Halaman akan di-reload setelah penghapusan berhasil
            } else {
                alert('Error deleting refill data');
            }
        })
        .catch(error => {
            alert('Failed to delete refill data: ' + error.message);
        });
}
```

Gambar 4.87 Fungsi deleteRefill()

Fungsi *deleteRefill()* pada Gambar 4.87 bertujuan untuk menghapus data *refill* atau pengisian ulang *earplug* berdasarkan *ID* tertentu yang dikirimkan sebagai parameter *refillId*. Sebelum proses penghapusan dijalankan, fungsi ini menampilkan dialog konfirmasi kepada pengguna melalui fungsi *confirm()* dengan pesan "*Are you sure you want to delete this refill record?*". Jika pengguna memilih untuk membatalkan, maka fungsi akan langsung dihentikan dengan perintah *return*.

Apabila pengguna mengonfirmasi untuk melanjutkan penghapusan, sistem akan mengirim permintaan menggunakan *fetch API* ke *file delete_refill.php* dengan metode *POST*. Data yang dikirim berupa *JSON* yang berisi *refill_id* yang akan dihapus. *Header Content-Type* diatur menjadi *'application/json'* untuk memastikan *server* mengenali format data yang dikirim.

Setelah permintaan dikirim, tanggapan dari *server* diolah dalam bentuk *JSON*. Jika tanggapan menunjukkan keberhasilan (*result.success bernilai true*), maka halaman akan diperbarui menggunakan *location.reload()* agar pengguna dapat melihat pembaruan data secara langsung. Namun, jika penghapusan gagal, maka sistem akan menampilkan peringatan melalui *alert* dengan pesan "*Error deleting refill data*". Sedangkan bila terjadi kegagalan jaringan atau kesalahan sistem lainnya, *catch block* akan menampilkan pesan kesalahan lengkap dengan informasi dari *error.message*.

Setelah fungsi *JavaScript deleteRefill()* diklik oleh pengguna dan permintaan penghapusan dikirim, maka *server-side script* yang akan menerima dan memproses permintaan tersebut adalah *file* PHP *delete_refill.php*. *File* inilah yang bertanggung jawab untuk mengeksekusi penghapusan data dari *database* secara langsung. Gambar 4.88 merupakan kode dari *delete_refill.php.*

```php
// Mengatur header sebelum output apa pun dikirim
header("Access-Control-Allow-Origin: *");
header("Content-Type: application/json; charset=UTF-8");

// Mengambil data JSON dari body request
$inputData = json_decode(file_get_contents('php://input'), true);

// Mendapatkan refill_id dari input JSON
$refillId = isset($inputData['refill_id']) ? intval($inputData['refill_id']) : 0;

if ($refillId == 0) {
    echo json_encode(['error' => 'Invalid refill ID']);
    exit;
}

try {
    // Cek apakah refill_id ada di database
    $checkQuery = "SELECT * FROM earplug_refills WHERE refill_id = :refill_id";
    $stmt = $pdo->prepare($checkQuery);
    $stmt->execute(['refill_id' => $refillId]);

    if ($stmt->rowCount() == 0) {
        // Jika tidak ditemukan, kirimkan respon error
        echo json_encode(['error' => 'Refill not found']);
        exit;
    }

    // Query untuk menghapus data berdasarkan refill_id
    $deleteQuery = "DELETE FROM earplug_refills WHERE refill_id = :refill_id";
    $stmt = $pdo->prepare($deleteQuery);
    $stmt->execute(['refill_id' => $refillId]);

    if ($stmt->rowCount() == 0) {
        // Jika tidak ditemukan, kirimkan respon error
        echo json_encode(['error' => 'Refill not found']);
        exit;
    }

    // Query untuk menghapus data berdasarkan refill_id
    $deleteQuery = "DELETE FROM earplug_refills WHERE refill_id = :refill_id";
    $stmt = $pdo->prepare($deleteQuery);
    $stmt->execute(['refill_id' => $refillId]);

    // Periksa apakah penghapusan berhasil
    if ($stmt->rowCount() > 0) {
        echo json_encode(['success' => true]);
    } else {
        echo json_encode(['error' => 'Failed to delete refill']);
    }
} catch (PDOException $e) {
    // Menangani error
    echo json_encode(['error' => $e->getMessage()]);
}
?>
```

Gambar 4.88 Kode delete_refill.php

Pertama, file ini memuat koneksi ke *database* melalui *include('dashboard_connection.php'),* yang memastikan bahwa seluruh perintah SQL berikutnya dapat dijalankan menggunakan objek koneksi $pdo. Kemudian, dua header dikirimkan yaitu *Access-Control-Allow-Origin:** agar dapat menerima permintaan dari domain mana pun (penting untuk komunikasi *cross-origin*), dan *Content-Type: application/json* yang menetapkan format balasan sebagai *JSON*.

Selanjutnya, skrip mengambil data yang dikirim dalam bentuk *JSON* melalui *file_get_contents('php://input')* dan mengubahnya menjadi *array associative* menggunakan *json_decode().* Dari data tersebut, *refill_id* diambil dan dikonversi ke *integer*. Jika tidak ditemukan atau bernilai nol, sistem langsung membalas dengan pesan kesalahan (*error*) bertuliskan *Invalid refill ID* dan menghentikan eksekusi dengan *exit*.

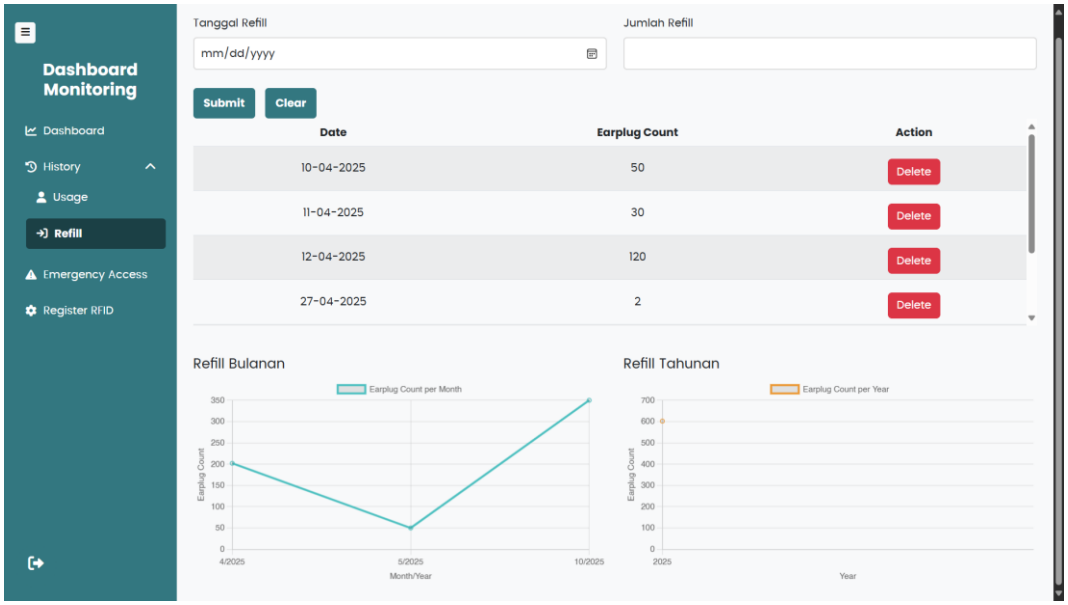
Jika *refill_id* valid, proses dilanjutkan dengan pemeriksaan apakah data tersebut benar-benar ada di dalam tabel *earplug_refills*. Pengecekan ini menggunakan *prepared statement* untuk menghindari serangan *SQL Injection*. Jika tidak ditemukan baris yang sesuai (*rowCount() == 0*), maka dikirimkan balasan *error* "*Refill not found*".

Apabila data ditemukan, maka sistem mengeksekusi perintah *SQL DELETE* untuk menghapus data berdasarkan *refill_id* yang diberikan. Setelah penghapusan dilakukan, sistem memeriksa kembali apakah ada baris yang terpengaruh oleh operasi ini. Jika ya, berarti penghapusan berhasil dan dikirimkan respons *{"success": true}.* Jika tidak, respons *error* dikirimkan dengan pesan "*Failed to delete refill*".

Seluruh operasi ini dibungkus dalam blok *try-catch* untuk menangani kemungkinan kesalahan yang terjadi selama proses, seperti kesalahan koneksi database atau perintah SQL yang gagal. Jika ada kesalahan, maka pesan dari pengecualian tersebut akan dikembalikan dalam bentuk *JSON* dengan kunci error.

Dengan pendekatan ini, *file delete_refill.php* tidak hanya menjalankan fungsi teknis penghapusan data, tetapi juga memberikan lapisan validasi dan keamanan, serta komunikasi yang jelas dalam format *API* berbasis *JSON* untuk integrasi

dengan *frontend*. Hasil pembuatan *Refill History Page* dapat dilihat pada Gambar 4.89.



Gambar 4.89 Refill History Page Pada Website

4.2.3.5 Pembuatan Program Emergency Access Page pada Website

Bagian *HTML* pada fitur *Emergency Access Page* ini dirancang untuk memungkinkan pengelola sistem melakukan pembaruan terhadap batas maksimal penggunaan harian (*daily tap limit*) dari pengguna sistem earplug berbasis *RFID*. Seluruh elemen dibungkus dalam sebuah * * dengan *id="emergencyAccessContent"* dan kelas *content-section*, yang berfungsi sebagai wadah utama konten *Emergency Access Page*.

```
<!-- Emergency Access Content -->
<div id="emergencyAccessContent" class="content-section">
    <h3>Emergency Access</h3>

    <!-- Form Update Tap Limit -->
    <form id="emergencyAccessForm" style="margin-top: 20px;">
        <div class="row">
            <div class="col-md-6">
                <div class="mb-3">
                    <label for="userID" class="form-label">Nama Pengguna</label>
                    <input type="text" class="form-control" id="userID" name="userID" readonly>
                </div>
            </div>
            <div class="col-md-6">
                <div class="mb-3">
                    <label for="newTapLimit" class="form-label">New Daily Tap Limit</label>
                    <input type="number" class="form-control" id="newTapLimit" name="newTapLimit" required min="1">
                </div>
            </div>
        </div>
        <div class="d-flex">
            <button type="submit" class="btn-submit me-2">Update Limit</button>
            <button type="button" id="clearButton" class="btn-clear">Clear</button>
        </div>
    </form>
    <p></p>
```

Gambar 4.90 Kode untuk Content Emergency Access Page

Gambar 4.90 menunjukkan kode untuk membangun *Content Emergency Access* Page. Pada bagian atas terdapat elemen <h3> bertuliskan *Emergency Access*, yang menjadi judul dari halaman ini. Di bawahnya, terdapat sebuah *form* yang diberi *id="emergencyAccessForm"*, yang akan digunakan untuk memperbarui data batas harian pengguna. Formulir ini memiliki dua *input* utama yang disusun dalam dua kolom. Kolom pertama berisi *input* teks dengan *id="userID"* yang hanya bisa dibaca (*readonly*) dan menampilkan nama pengguna yang dipilih dari tabel. Kolom kedua berisi *input* angka *(type="number")* dengan *id="newTapLimit"* yang memungkinkan pengelola mengatur ulang batas *tap* harian, dengan nilai minimal yang diizinkan adalah 1 (*min="1"*).

Setelah isian form, terdapat dua tombol yaitu tombol *submit* dengan kelas *btn-submit* untuk mengirim pembaruan ke sistem, dan tombol *clear* dengan *id="clearButton"* yang berfungsi untuk mengosongkan kolom *form*, memudahkan pengelola saat ingin berpindah ke pengguna lain.

```
<!-- Tabel Data Pengguna -->
<table id="userTable" class="table">
    <thead>
        <tr>
            <th>Nama</th>
            <th>RFID Tag</th>
            <th>Daily Tap Limit</th>
            <th>Status</th>
            <th>Action</th>
        </tr>
    </thead>
    <tbody>
        <!-- Data dari server akan dimasukkan di sini melalui JavaScript -->
    </tbody>
</table>
</div>
```

Gambar 4.91 Kode untuk Tabel pada Emergency Access Page

Selanjutnya, terdapat tabel HTML seperti pada Gambar 4.91 dengan *id="userTable"* yang akan menampilkan daftar pengguna sistem. Tabel ini terdiri dari lima kolom yaitu Nama, *RFID Tag*, *Daily Tap Limit*, *Status*, dan *Action*. Kolom *Action* nantinya akan diisi dengan tombol-tombol aksi (seperti pilih/edit) melalui *JavaScript*. Data dalam tabel ini tidak ditulis secara statis dalam HTML, melainkan akan dimuat secara dinamis dari *server* menggunakan skrip *JavaScript*.

```
//Proses Update Limit
document.getElementById('emergencyAccessForm').addEventListener('submit', function (e) {
    e.preventDefault(); // Mencegah form reload halaman

    // Ambil data dari form
    const userID = document.getElementById('userID').value;  // Nama Pengguna
    const newTapLimit = document.getElementById('newTapLimit').value;  // Tap Limit Baru

    // Kirim data ke PHP melalui AJAX
    fetch('update_tap_limit.php', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded',
        },
        body: `userID=${encodeURIComponent(userID)}&newTapLimit=${encodeURIComponent(newTapLimit)}`
    })
    .then((response) => response.json())
    .then((data) => {
        alert(data.message); // Tampilkan pesan dari server
        if (data.success) {
            // Reset form jika sukses
            document.getElementById('emergencyAccessForm').reset();

            // Reload halaman untuk memuat data terbaru
            window.location.reload();
        }
    })
    .catch((error) => {
        alert('An error occurred: ' + error.message);
    });
});
```

Gambar 4.92 Kode untuk memproses Update Limit Harian

Pada Gambar 4.92 ditampilkan kode untuk memproses Update Limit Harian. Bagian *JavaScript* ini berfungsi untuk menangani proses pembaruan *daily tap limit* pada halaman *Emergency Access*. Proses dimulai ketika pengguna menekan tombol *submit* pada *form* dengan *id="emergencyAccessForm"*. *Event listener addEventListener('submit', function (e) {...})* digunakan untuk menangkap peristiwa tersebut dan menjalankan fungsi yang didefinisikan.

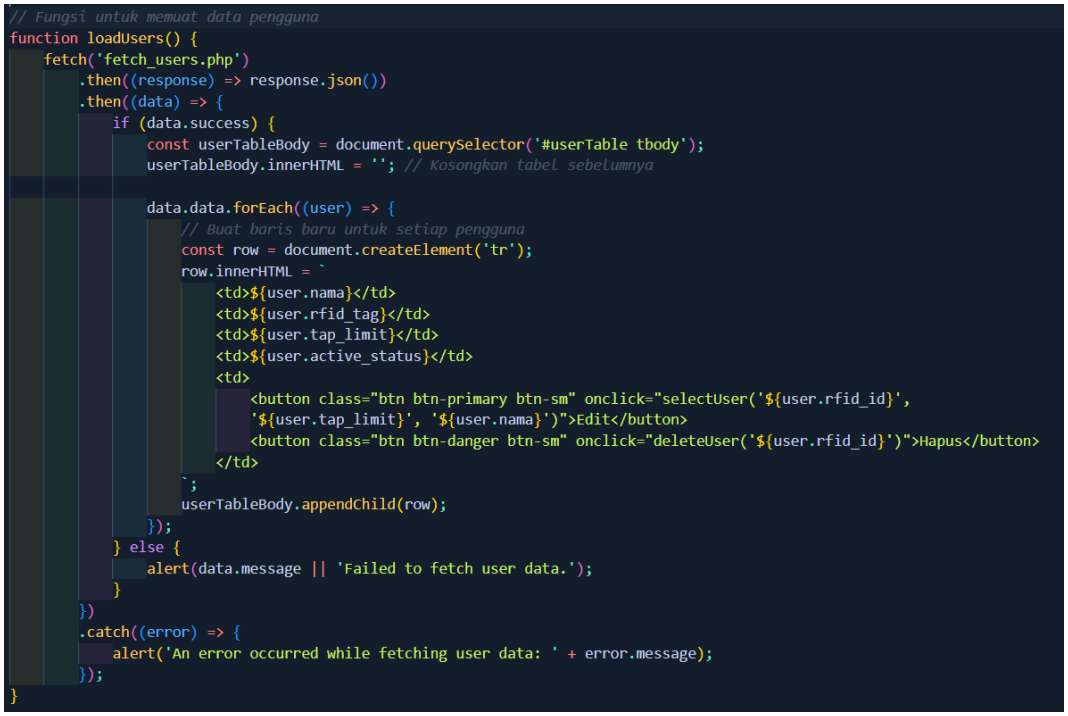
Langkah pertama dalam fungsi ini adalah memanggil *e.preventDefault();,* yang bertujuan untuk mencegah perilaku bawaan *form* yaitu me-*reload* halaman saat disubmit. Hal ini penting agar proses pengiriman data bisa dilakukan secara *asynchronous* melalui *AJAX* tanpa harus merefresh seluruh tampilan.

Selanjutnya, *JavaScript* mengambil nilai yang pengguna *input* dari elemen *input* pada *form*. Nilai *userID* diambil dari elemen *#userID*, yang sebelumnya sudah di-*populate* saat pengguna dipilih dari tabel, sedangkan nilai *newTapLimit* diambil dari *input* angka yang ditetapkan sebagai batas *tap* baru.

Setelah kedua nilai diperoleh, proses pengiriman data ke *server-side* dimulai dengan menggunakan *fetch API*. Permintaan dikirim ke file *update_tap_limit.php* dengan metode *POST*, dan tipe konten dikirim dalam bentuk *application/x-www-form-urlencoded*, format umum yang digunakan oleh *form* HTML standar. Data proses menggunakan *encodeURIComponent()* untuk memastikan bahwa karakter khusus dalam nilai tidak menyebabkan *error* saat diterima di sisi *server*.

Ketika *server* merespons, data dikonversi menjadi *JSON* menggunakan *response.json().* Jika permintaan berhasil dan *server* mengirimkan pesan dengan status sukses (*data.success*), maka sebuah notifikasi akan ditampilkan menggunakan *alert(data.message), form* akan di-*reset*, dan halaman akan di-*reload* agar perubahan batas *tap* dapat langsung terlihat oleh pengguna.

Jika terjadi kesalahan dalam proses pengiriman atau pada saat mendapatkan respon dari *server*, maka akan muncul peringatan (*alert*) yang menunjukkan pesan kesalahan melalui *blok catch*.



```
// Fungsi untuk memuat data pengguna
function loadUsers() {
    fetch('fetch_users.php')
        .then((response) => response.json())
        .then((data) => {
            if (data.success) {
                const userTableBody = document.querySelector('#userTable tbody');
                userTableBody.innerHTML = ''; // Kosongkan tabel sebelumnya

                data.data.forEach((user) => {
                    // Buat baris baru untuk setiap pengguna
                    const row = document.createElement('tr');
                    row.innerHTML = `
                        <td>${user.nama}</td>
                        <td>${user.rfid_tag}</td>
                        <td>${user.tap_limit}</td>
                        <td>${user.active_status}</td>
                        <td>
                            <button class="btn btn-primary btn-sm" onclick="selectUser('${user.rfid_id}',
                            '${user.tap_limit}', '${user.nama}')">Edit</button>
                            <button class="btn btn-danger btn-sm" onclick="deleteUser('${user.rfid_id}')">Hapus</button>
                        </td>
                    `;
                    userTableBody.appendChild(row);
                });
            } else {
                alert(data.message || 'Failed to fetch user data.');
            }
        })
        .catch((error) => {
            alert('An error occurred while fetching user data: ' + error.message);
        });
}
```

Gambar 4.93 Kode untuk Fungsi loadUsers()

Pada Gambar 4.93 menunjukkan kode untuk fungsi *loadUsers()*. Kode *JavaScript* ini bertugas untuk memuat data pengguna dari *server* dan menampilkannya ke dalam tabel yang ada di halaman *Emergency Access*. Fungsi ini bernama *loadUsers()* dan merupakan bagian penting dari mekanisme *dynamic content rendering*, di mana data ditampilkan berdasarkan hasil permintaan ke *server* tanpa perlu me-*reload* halaman secara manual.

Fungsi dimulai dengan memanggil *fetch('fetch_users.php'),* yaitu permintaan ke file *PHP* bernama *fetch_users.php* yang bertugas menyediakan data pengguna dalam format *JSON*. Ketika *response* diterima, data diubah menjadi format *JavaScript Object* melalui *response.json()*.

Selanjutnya, program melakukan pengecekan terhadap *data.success*. Jika nilai ini *true*, artinya data berhasil diperoleh. Kemudian, elemen *tbody* dari tabel dengan *id="userTable"* diakses dan isinya dikosongkan terlebih dahulu menggunakan *innerHTML = ''* agar tidak terjadi penumpukan data saat pemuatan ulang.

Kemudian, untuk setiap elemen pengguna dalam *data.data*, dilakukan perulangan menggunakan *forEach()*. Setiap data pengguna digunakan untuk membuat baris baru pada tabel (<tr>), yang berisi kolom-kolom seperti nama (*user.nama*), *RFID tag* (*user.rfid_tag*), *tap limit* (*user.tap_limit*), dan *status aktif* (*user.active_status*). Kolom terakhir berisi dua tombol aksi yaitu tombol Edit dan tombol Hapus.

Tombol *Edit* memiliki atribut *onclick* yang akan memicu fungsi *selectUser()* dengan parameter *rfid_id, tap_limit*, dan nama. Fungsi ini akan mengisi *form Emergency Access* dengan data pengguna tersebut agar dapat diedit. Sementara itu, tombol *Hapus* akan memanggil fungsi *deleteUser()* untuk menghapus data pengguna berdasarkan *rfid_id.*

Jika terjadi kesalahan saat memuat data, baik karena *server* tidak merespons atau karena struktur data tidak sesuai, maka akan muncul pesan *alert* yang menjelaskan kesalahan tersebut. *Blok catch()* di akhir fungsi akan menangani kemungkinan kesalahan jaringan atau kesalahan lainnya, dan akan menampilkan pesan kesalahan melalui *alert()*.

```
function deleteUser(rfid_id) {
    // Tanyakan konfirmasi sebelum menghapus
    if (confirm('Are you sure you want to delete this user?')) {
        fetch('delete_user.php', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/x-www-form-urlencoded',
            },
            body: `rfid_id=${rfid_id}`,
        })
        .then(response => response.json())
        .then(data => {
            if (data.success) {
                alert('User deleted successfully!');
                loadUsers(); // Reload data setelah menghapus
            } else {
                alert('Failed to delete user: ' + (data.message || 'Unknown error'));
            }
        })
        .catch(error => {
            alert('An error occurred while deleting the user: ' + error.message);
        });
    }
}
```

Gambar 4.94 Kode untuk fungsi deleteUser()

Pada Gambar 4.94 menunjukkan Fungsi *deleteUser(rfid_id)*. Kode ini digunakan untuk menghapus data pengguna berdasarkan *RFID ID*. Ketika fungsi dipanggil, sistem terlebih dahulu menampilkan *dialog box* konfirmasi menggunakan *confirm()*. Tujuan dari langkah ini adalah untuk memastikan bahwa pengguna benar-benar ingin melakukan penghapusan data, sehingga dapat menghindari tindakan yang tidak disengaja. Jika pengguna memilih *cancel*, maka proses akan langsung dihentikan dan tidak ada aksi yang dilakukan.

Namun jika pengguna memilih untuk melanjutkan, fungsi ini akan mengirimkan permintaan ke *server* melalui metode *HTTP POST* ke file *delete_user.php*. Data yang dikirim dalam permintaan ini adalah *parameter rfid_id*, yang diformat menggunakan *Content-Type application/x-www-form-urlencoded*. Setelah permintaan dikirim, sistem akan menunggu respons dari *server*. Respons tersebut kemudian diubah ke format *JSON* agar bisa diproses lebih lanjut.

Jika hasil dari *server* menunjukkan bahwa proses berhasil (yaitu nilai *success* bernilai *true*), maka sistem akan menampilkan *alert* berisi pesan bahwa pengguna telah berhasil dihapus. Selanjutnya, fungsi *loadUsers()* akan dipanggil kembali untuk memperbarui tabel pengguna di halaman, sehingga pengguna yang dihapus tidak lagi ditampilkan. Namun jika terjadi kegagalan dalam proses penghapusan (misalnya karena *RFID ID* tidak valid), maka pesan kesalahan akan ditampilkan kepada pengguna.

Terakhir, jika terjadi gangguan seperti masalah jaringan atau kesalahan pada sisi *server*, blok *.catch()* akan menangani *error* tersebut dan menampilkan *alert* yang menjelaskan kesalahan yang terjadi.

```
// Fungsi untuk mengisi data ke form ketika tombol "Edit" diklik
function selectUser(userID, tapLimit, userName) {
    document.getElementById('userID').value = userName;
    document.getElementById('newTapLimit').value = tapLimit;
}
```

Gambar 4.95 Kode untuk Tombol Edit

Pada Gambar 4.95 ditunjukkan kode untuk pemrosesan tombol edit. Fungsi *selectUser(userID, tapLimit, userName)* digunakan untuk *mengisi otomatis* data ke

dalam *form* ketika pengguna mengklik tombol *"Edit"* pada salah satu baris di tabel. Fungsi ini bekerja dengan menangkap tiga parameter yaitu *userID, tapLimit*, dan *userName*, yang masing-masing mewakili *ID* pengguna (dalam hal ini adalah *RFID ID*), *batas tap harian baru*, dan nama pengguna.

Di dalam fungsi, nilai userName langsung dimasukkan ke dalam elemen input dengan *id userID*. Meskipun nama *id*-nya *userID*, *input* ini digunakan untuk menampilkan nama pengguna yang sedang dipilih. Kemudian, nilai *tapLimit* dimasukkan ke dalam *input field newTapLimit*, yang berfungsi untuk mengatur atau mengubah batas harian *tap* yang baru.

```php
<?php
// Include file koneksi database
require_once 'dashboard_connection.php'; // Ganti dengan nama file koneksi database Anda

// Cek apakah data form sudah dikirimkan
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    // Ambil data dari form
    $userID = isset($_POST['userID']) ? trim($_POST['userID']) : ''; // Nama Pengguna
    $newTapLimit = isset($_POST['newTapLimit']) ? intval($_POST['newTapLimit']) : 0; // Tap Limit Baru

    // Validasi data
    if (empty($userID) || $newTapLimit < 1) {
        echo json_encode([
            'success' => false,
            'message' => 'Invalid input data. Please fill out all fields correctly.'
        ]);
        exit;
    }

    try {
        // Koneksi ke database sudah dilakukan di 'dashboard_connection.php', kita langsung gunakan $pdo

        // Cari rfid_id berdasarkan nama pengguna (userID)
        $query = "SELECT rfid_id FROM rfid_users WHERE nama = :userID";
        $stmt = $pdo->prepare($query);
        $stmt->bindParam(':userID', $userID, PDO::PARAM_STR);
        $stmt->execute();

        // Jika user ditemukan
        if ($stmt->rowCount() > 0) {
            $user = $stmt->fetch(PDO::FETCH_ASSOC);
            $rfid_id = $user['rfid_id'];
```

```php
        // Update tap_limit berdasarkan rfid_id
        $updateQuery = "UPDATE rfid_users SET tap_limit = :newTapLimit WHERE rfid_id = :rfid_id";
        $updateStmt = $pdo->prepare($updateQuery);
        $updateStmt->bindParam(':newTapLimit', $newTapLimit, PDO::PARAM_INT);
        $updateStmt->bindParam(':rfid_id', $rfid_id, PDO::PARAM_INT);

        if ($updateStmt->execute()) {
            echo json_encode([
                'success' => true,
                'message' => 'Tap limit updated successfully.'
            ]);
        } else {
            echo json_encode([
                'success' => false,
                'message' => 'Failed to update tap limit. Please try again.'
            ]);
        }
    } else {
        echo json_encode([
            'success' => false,
            'message' => 'User not found.'
        ]);
    }

} catch (PDOException $e) {
    echo json_encode([
        'success' => false,
        'message' => 'Database error: ' . $e->getMessage()
    ]);
}
} else {
    echo json_encode([
        'success' => false,
        'message' => 'Invalid request method.'
    ]);
}
```

Gambar 4.96 Kode update_tap_limit.php

Gambar 4.96 merupakan kode *update_tap_limit.*php. Keseluruhan kode tersebut merupakan skrip *backend* berbasis *PHP* yang digunakan untuk memperbarui batas *tap* (atau *tap limit*) dari pengguna berdasarkan nama mereka. Proses dimulai dengan menyertakan koneksi ke *database* melalui *require_once 'dashboard_connection.php'*, yang menyimpan objek koneksi *PDO* bernama $pdo.

Selanjutnya, skrip memeriksa apakah permintaan yang masuk menggunakan metode *POST*. Jika iya, maka data *userID* (nama pengguna) dan *newTapLimit* (batas *tap* baru) diambil dari *form* dan divalidasi. Validasi memastikan bahwa nama tidak kosong dan batas *tap* lebih dari 0. Jika tidak valid, skrip mengembalikan respon *JSON* yang menyatakan kesalahan *input*.

Jika data valid, maka dilanjutkan ke tahap pencarian pengguna di tabel *rfid_users* menggunakan *prepared statement* untuk mencegah *SQL injection*. Skrip

mencari *rfid_id* yang sesuai dengan nama pengguna. Jika ditemukan *($stmt->rowCount() > 0),* maka *rfid_id* diambil dan digunakan untuk memperbarui nilai *tap_limit* melalui perintah *UPDATE*.

Proses pembaruan juga dilakukan secara aman menggunakan *prepared statement*, di mana nilai *:newTapLimit* dan *:rfid_id di-bind* ke variabel yang sesuai. Jika perintah *UPDATE* berhasil dieksekusi, sistem mengembalikan respon *JSON* yang menyatakan keberhasilan dengan pesan *"Tap limit updated successfully."* Sebaliknya, jika gagal, akan dikembalikan pesan bahwa pembaruan gagal.

Apabila pengguna tidak ditemukan berdasarkan nama yang diberikan, maka sistem akan mengirimkan respon *JSON* bahwa pengguna tidak ditemukan. Di sisi lain, jika terjadi kesalahan saat mengakses *database*, misalnya karena koneksi bermasalah atau *query* tidak valid, maka akan ditangkap oleh *blok catch* dan dikembalikan pesan kesalahan yang berisi detail dari objek *exception PDOException*.

Jika permintaan tidak menggunakan metode *POST*, maka langsung dikembalikan pesan *"Invalid request method."* Hal ini menjaga agar skrip hanya dapat diakses dengan cara yang sesuai dan aman, yaitu melalui *form submission* dengan metode *POST*.

```php
<?php
// Include file koneksi database
require_once 'dashboard_connection.php'; // Ganti dengan nama file koneksi database Anda

// Header JSON untuk memastikan respons JSON dikirim
header('Content-Type: application/json');

try {
    // Query untuk mengambil data pengguna dari tabel rfid_users
    $query = "SELECT rfid_id, no_induk, rfid_tag, nama, divisi, tap_limit, active_status FROM rfid_users";
    $stmt = $pdo->prepare($query);
    $stmt->execute();

    // Ambil semua data pengguna
    $users = $stmt->fetchAll(PDO::FETCH_ASSOC);

    // Mengirim respons JSON dengan data pengguna
    echo json_encode([
        'success' => true,
        'data' => $users,
    ]);
} catch (PDOException $e) {
    // Menangani error jika terjadi kesalahan dalam pengambilan data
    echo json_encode([
        'success' => false,
        'message' => 'Database error: ' . $e->getMessage(),
    ]);
    exit;
}
?>
```

Gambar 4.97 Kode fetch_user.php

Pada Gambar 4.97 ditampilkan kode pada *file fetch_user.php.* Kode *PHP* tersebut digunakan untuk menampilkan data seluruh pengguna dari tabel *rfid_users* dalam format *JSON*. Pertama, kode menyertakan *file* koneksi *database* menggunakan *require_once 'dashboard_connection.php'* untuk memastikan bahwa koneksi ke *database* telah tersedia melalui objek *PDO*. Selanjutnya, perintah *header('Content-Type: application/json');* digunakan untuk menetapkan bahwa *output* dari *file* ini adalah dalam format *JSON*, sehingga dapat dibaca dengan benar oleh sisi *frontend* seperti aplikasi *JavaScript* atau *framework web*.

Di dalam *blok try*, sistem menjalankan *query* SQL untuk mengambil beberapa kolom yaitu *rfid_id*, *no_induk*, *rfid_tag*, *nama*, *divisi*, *tap_limit*, dan *active_status* dari tabel *rfid_users*. Untuk alasan keamanan dan performa, *query* ini dieksekusi menggunakan *prepared statement* melalui *$pdo->prepare($query);* dan kemudian dijalankan dengan *$stmt->execute();*. Setelah itu, hasil data dari *query* diambil menggunakan *$stmt->fetchAll(PDO::FETCH_ASSOC);* yang menghasilkan *array* asosiatif dan disimpan dalam variabel *$users*.

Apabila proses pengambilan data berhasil, maka hasilnya dikirim ke klien dalam bentuk *JSON* dengan struktur yang berisi *key* 'success' => true dan *'data' => $users*, yang mengindikasikan bahwa permintaan berhasil dan data dapat digunakan di sisi *frontend*. Namun, jika terjadi kesalahan selama proses, seperti kesalahan koneksi atau eksekusi *query*, maka akan ditangani dalam blok catch. Di dalamnya, sistem akan mengembalikan *response JSON* dengan *'success' => false* dan *message* berisi penjelasan kesalahan berdasarkan pesan dari *exception* yang ditangkap, dalam hal ini *PDOException*.

```php
<?php
// Header JSON
header('Content-Type: application/json');

// Include file koneksi
require_once 'dashboard_connection.php';

try {
    // Ambil data dari request POST
    $rfid_id = $_POST['rfid_id'] ?? '';

    // Validasi input
    if (empty($rfid_id)) {
        echo json_encode([
            'success' => false,
            'message' => 'Parameter rfid_id tidak ditemukan',
        ]);
        exit;
    }

    // Query hapus berdasarkan rfid_id
    $query = "DELETE FROM rfid_users WHERE rfid_id = :rfid_id";
    $stmt = $pdo->prepare($query);
    $stmt->bindParam(':rfid_id', $rfid_id, PDO::PARAM_INT);
```

Gambar 4.98 Logika Awal Penghapusan Data Melalui delete_user.php

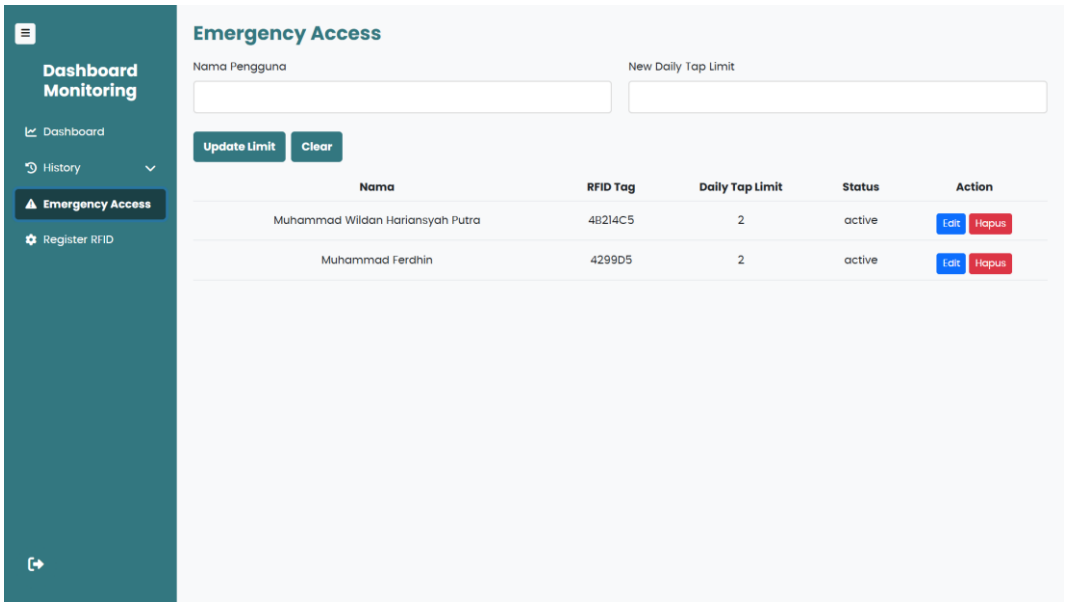
Pada Gambar 4.98 ditampilkan logika awal pada *file delete_user.php.* Kode *PHP* tersebut merupakan bagian awal dari proses penghapusan data pengguna berdasarkan *rfid_id* melalui metode *POST*, dan dirancang untuk memberikan *response* dalam format *JSON*. Bagian pertama dari kode, yaitu *header('Content-Type: application/json');,* digunakan untuk memberi tahu *client* bahwa data balasan dari *server* akan berupa *JSON*, yang sangat penting terutama ketika proses ini dijalankan melalui permintaan *AJAX* di sisi *frontend*.

Selanjutnya, file koneksi ke *database* dimuat dengan *require_once 'dashboard_connection.php';,* sehingga objek *PDO* yang dibutuhkan untuk eksekusi *query* sudah tersedia. Kemudian, data *rfid_id* diambil dari permintaan *POST* menggunakan *$rfid_id = $_POST['rfid_id'] ?? '';*. Penulisan ini berarti jika

parameter rfid_id tidak dikirim, maka secara default nilainya akan menjadi string kosong.

Untuk memastikan data yang dikirim valid, dilakukan proses validasi melalui *if (empty($rfid_id))*. Jika *rfid_id* kosong atau tidak ada, sistem langsung mengembalikan *response JSON* dengan status *success false* dan pesan bahwa parameter tidak ditemukan, lalu menghentikan eksekusi menggunakan exit.

Jika input valid, maka proses dilanjutkan dengan pembuatan *query* penghapusan menggunakan *DELETE FROM rfid_users WHERE rfid_id = :rfid_id*, yang artinya akan menghapus baris dalam tabel *rfid_users* berdasarkan nilai *rfid_id*. *Query* ini dieksekusi menggunakan *prepared statement* untuk alasan keamanan dan efisiensi. Nilai *rfid_id* diikat ke dalam *query* menggunakan *$stmt->bindParam(':rfid_id', $rfid_id, PDO::PARAM_INT);*, yang memastikan bahwa parameter dikirim dalam bentuk bilangan bulat agar sesuai dengan tipe data kolom di *database*. Pada Gambar 4.99 ditunjukkan tampilan *Emergency Access Page* pada *website monitoring* yang dibangun.



Gambar 4.99 Tampilan Emergency Access Page pada Website

4.2.3.6 Pembuatan Program Register RFID Page pada Website

Pada tahap ini dilakukan pembuatan *Register RFID Page* yang berfungsi untuk mendaftarkan *tag* kartu RFID agar bisa mendapat akses pengambilan *earplug*. Pada Gambar 4.100 ditampilkan kode HTML yang digunakan untuk membentuk *interface* pada *Register RFID Page*.

```
<!-- Register RFID Content -->
<div id="registerRFIDContent" class="content-section">
    <h3>Register RFID</h3>
    <form id="registerRFIDForm" method="POST" action="register_rfid.php">
        <div class="row">
            <div class="col-md-6">
                <div class="mb-3">
                    <label for="rfidName" class="form-label">Nama</label>
                    <input type="text" class="form-control" id="rfidName" name="rfidName" required>
                </div>
            </div>
            <div class="col-md-6">
                <div class="mb-3">
                    <label for="rfidDivision" class="form-label">Department</label>
                    <input type="text" class="form-control" id="rfidDivision" name="rfidDivision" required>
                </div>
            </div>
        </div>
        <div class="row">
            <div class="col-md-6">
                <div class="mb-3">
                    <label for="rfidNoInduk" class="form-label">No Induk</label>
                    <input type="text" class="form-control" id="rfidNoInduk" name="rfidNoInduk" required>
                </div>
            </div>
            <div class="col-md-6">
                <div class="mb-3">
                    <label for="rfidTag" class="form-label">RFID Tag</label>
                    <input type="text" class="form-control" id="rfidTag" name="rfidTag" required>
                </div>
            </div>
        </div>
        <div class="form-actions">
            <button type="submit" class="btn-submit">Register RFID</button>
            <button type="button" class="btn-clear" id="clearFormButton">Clear</button>
        </div>
```

Gambar 4.100 Bagian HTML Register RFID Page

Bagian *HTML* ini merupakan *interface* pengguna untuk fitur pendaftaran kartu *RFID* baru dalam sistem. Elemen utama dari tampilan ini adalah sebuah *div* dengan atribut *id="registerRFIDContent"* dan kelas *content-section*, yang menandakan bahwa bagian ini adalah salah satu *section* dari halaman utama. Di dalamnya terdapat elemen <h3> yang memberikan judul halaman berupa *Register RFID*, sebagai penanda bahwa pengguna dapat mendaftarkan data kartu *RFID* baru melalui *form* ini.

Formulir ini menggunakan *tag* <form> dengan atribut *id="registerRFIDForm",* metode *POST*, dan *action* yang mengarah ke *file register_rfid.php*. Artinya, saat pengguna mengisi dan mengirimkan formulir, data akan dikirim ke *server* melalui metode *POST* dan diproses oleh *file PHP* tersebut. Tampilan *form* dibagi menjadi dua baris (*row*), yang masing-masing terdiri dari dua kolom menggunakan kelas *Bootstrap col-md-6*, untuk memastikan tampilan tetap rapi dan responsif di berbagai ukuran layar.

Pada baris pertama, terdapat dua *input*, yaitu untuk *Nama* dan *Department*. Input *Nama* dibuat menggunakan elemen *<input type="text">* dengan *id="rfidName"* dan *name="rfidName",* serta bersifat *required* agar tidak boleh kosong. Begitu pula dengan input *Department*, yang menggunakan *id="rfidDivision"* dan *name="rfidDivision"*. Keduanya digunakan untuk mencatat identitas pengguna.

Baris kedua memuat *input* untuk *No Induk* dan *RFID Tag*. *No Induk* merupakan nomor identifikasi unik dari pengguna yang diinput melalui *id="rfidNoInduk"* dan *name="rfidNoInduk"*, sedangkan *RFID Tag* adalah kode atau *tag* unik dari kartu *RFID* itu sendiri, yang diisi melalui *id="rfidTag"* dan *name="rfidTag"*. Semua *input* ini bersifat wajib diisi (*required*).

Di akhir *form* terdapat dua tombol aksi (*action buttons*). Tombol pertama adalah tombol untuk mengirimkan form (type="submit") dengan label *Register RFID*, yang akan menjalankan proses pengiriman data ke *backend*. Tombol kedua *(type="button")* berlabel *Clear* dan memiliki *id="clearFormButton",* digunakan untuk mengosongkan seluruh isian *form*. Tombol ini dikaitkan dengan fungsi *JavaScript* untuk menjalankan perintah pembersihan isian.

```
// Clear form when the "Clear" button is clicked
document.getElementById("clearFormButton").addEventListener("click", function () {
document.getElementById("registerRFIDForm").reset();
});
```

Gambar 4.101 Kode untuk Mengosongkan Form pada Register RFID Page

Kode JavaScript pada Gambar 4.101 digunakan untuk menangani aksi dari tombol *Clear* pada halaman *Register RFID*. Baris pertama kode

document.getElementById("clearFormButton") berfungsi untuk mengambil elemen tombol yang memiliki atribut *id* bernama *clearFormButton*. Selanjutnya, metode *addEventListener("click", function () { ... })* digunakan untuk menambahkan *event listener* terhadap elemen tersebut. Artinya, ketika tombol *Clear* diklik oleh pengguna, fungsi di dalamnya akan dijalankan. Di dalam fungsi tersebut terdapat perintah *document.getElementById("registerRFIDForm").reset();* yang berfungsi untuk mereset atau mengosongkan seluruh isian pada *form* dengan *id registerRFIDForm*. Metode *.reset()* ini akan mengembalikan semua nilai *input* dalam *form* ke kondisi awal, yaitu kosong seperti saat pertama kali halaman dimuat. Dengan demikian, kode ini memberikan kemudahan bagi pengguna untuk menghapus semua isian *form* secara instan tanpa perlu melakukannya satu per satu secara manual.

```php
<?php
require 'dashboard_connection.php';

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    // Retrieve form data
    $name = $_POST['rfidName'];
    $division = $_POST['rfidDivision'];
    $no_induk = $_POST['rfidNoInduk'];
    $rfid_tag = $_POST['rfidTag'];

    // Validate input (basic validation)
    if (empty($name) || empty($division) || empty($no_induk) || empty($rfid_tag)) {
        // Redirect back with error message in query string
        header("Location: http://localhost/TA/Website%20(1)/Dashboard.html?status=error&message=All%20fields%20are%20required.");
        exit;
    }

    try {
        // Insert data into database
        $stmt = $pdo->prepare("INSERT INTO rfid_users (no_induk, rfid_tag, nama, divisi, tap_limit, active_status, created_at)
                               VALUES (:no_induk, :rfid_tag, :nama, :divisi, 2, 'active', NOW())");
        $stmt->execute([
            ':no_induk' => $no_induk,
            ':rfid_tag' => $rfid_tag,
            ':nama' => $name,
            ':divisi' => $division
        ]);

        // Redirect back with success message in query string
        header("Location: http://localhost/TA/Website%20(1)/Dashboard.html?status=success&message=RFID%20registered%20successfully.");
        exit;
    } catch (PDOException $e) {
        // Redirect back with error message in query string
        header("Location: http://localhost/TA/Website%20(1)/Dashboard.html?status=error&message=" . urlencode($e->getMessage()));
        exit;
    }
}
```

Gambar 4.102 Kode register_rfid.php

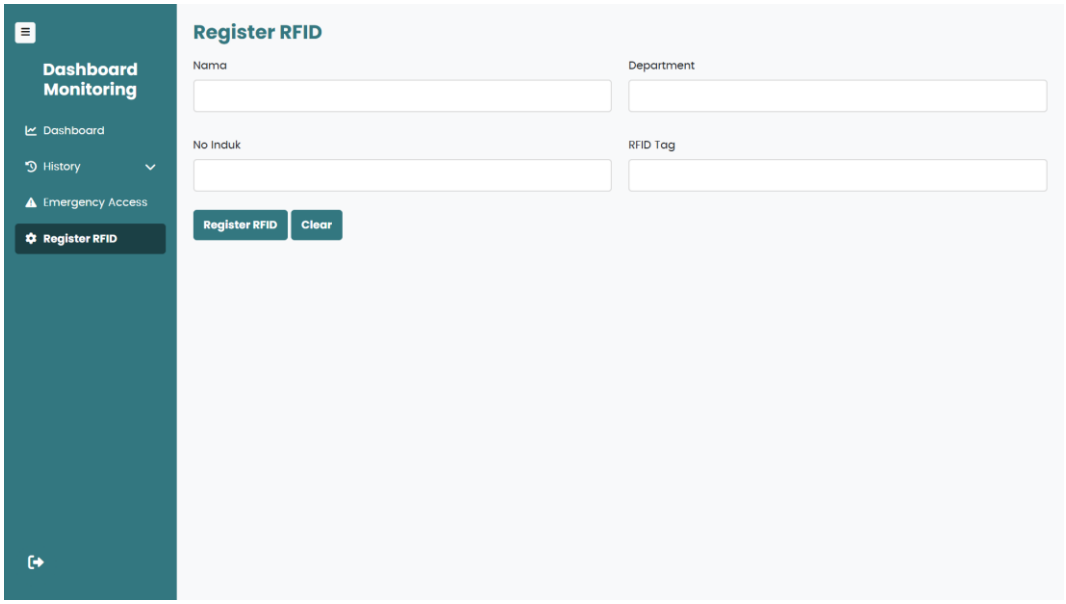
Pada Gambar 4.102 ditampilkan kode pada *file register_rfid.php.* Kode *PHP* tersebut digunakan untuk memproses *form* pendaftaran *RFID* dari halaman *Register RFID Page*. Pertama, *file* koneksi ke *database* yang bernama *dashboard_connection.php* di-*include* menggunakan perintah *require*. Kemudian,

dilakukan pengecekan apakah permintaan yang masuk adalah metode *POST* dengan *$_SERVER['REQUEST_METHOD'] == 'POST'*, karena data dari *form* dikirim melalui metode tersebut.

Selanjutnya, data yang dikirim dari *form* diambil satu per satu menggunakan *$_POST*, yaitu *rfidName*, *rfidDivision*, *rfidNoInduk*, dan *rfidTag*, yang masing-masing disimpan dalam variabel *PHP*. Setelah itu, dilakukan validasi dasar menggunakan fungsi *empty()* untuk memastikan semua *input field* telah diisi. Jika ada yang kosong, maka pengguna akan langsung diarahkan kembali ke halaman utama *dashboard* dengan *query string* berisi status *error* dan pesan bahwa semua kolom wajib diisi.

Jika semua data telah valid, maka kode akan masuk ke blok *try*, yang digunakan untuk menangani kemungkinan kesalahan saat melakukan operasi ke *database*. Di dalam blok ini, disiapkan *prepared statement* untuk memasukkan data ke dalam tabel *rfid_users*. Nilai *tap_limit* langsung diatur sebesar 2, dan *active_status* diset ke *'active'*. Kolom *created_at* diisi dengan fungsi *MySQL NOW()* untuk mencatat waktu saat data dimasukkan.

Setelah proses *insert* berhasil, pengguna akan diarahkan kembali ke halaman *Dashboard* dengan status *success* dan pesan bahwa *RFID* berhasil diregistrasi. Namun, jika terjadi kesalahan selama eksekusi, misalnya karena duplikasi atau masalah *database* lainnya, maka proses masuk ke blok *catch*, dan pengguna juga diarahkan kembali ke halaman *Dashboard*, namun kali ini dengan status *error* serta pesan kesalahan yang telah di-*encode* agar aman ditampilkan di *URL*. Pada Gambar 4.103 ditampilkan *interface Register RFID Page* pada *website*.



Gambar 4.103 Tampilan Register RFID Page pada Website

4.2.4 Pembuatan Program Aplikasi Mobile

Pada tahap ini, dilakukan proses pembuatan aplikasi *mobile* bernama *Earmate* yang berfungsi sebagai antarmuka pengguna untuk melakukan pemantauan dan pengelolaan sistem dispenser *earplug* otomatis.

Gambar 4.104 Logo Earmate

Pada Gambar 4.104 ditampilkan logo aplikasi *mobile Earmate*. Aplikasi ini dirancang agar memiliki fitur yang sama dengan versi *website*, seperti *monitoring* data sisa *earplug*, pencatatan histori penggunaan harian, bulanan, dan tahunan, serta menu untuk melakukan *refill*, registrasi *RFID tag*, hingga penghapusan data *RFID*. Aplikasi ini dikembangkan menggunakan *framework* Flutter dan ditulis menggunakan bahasa pemrograman Dart dengan *code editor* Visual Studio Code.

Sebagai *backend*, aplikasi *Earmate* terhubung dengan *database MySQL* melalui layanan *PHP script* yang berjalan di server *localhost*. Proses pengambilan dan pengiriman data dilakukan secara *fetch* menggunakan protokol HTTP, dengan metode *POST* dan *GET* untuk pertukaran informasi. Aplikasi ini telah dilengkapi dengan fitur *auto-refresh* agar data pada tampilan dapat diperbarui secara berkala tanpa harus dimuat ulang secara manual oleh pengguna. Saat ini, akses terhadap aplikasi hanya dibatasi untuk pengguna dengan hak akses *admin* guna memastikan kontrol penuh terhadap proses distribusi dan manajemen data.

4.2.4.1 Pembuatan Program Welcome Page pada Aplikasi Mobile

Halaman ini dirancang dengan mempertimbangkan tampilan yang menarik, informatif, dan mudah digunakan. Pengembangan dilakukan menggunakan bahasa pemrograman Dart dengan bantuan *framework* Flutter, di mana seluruh kode diletakkan dalam *file welcome_screen.dart*. Komponen utama halaman ini dibangun menggunakan *widget Scaffold*, yang menjadi wadah utama untuk seluruh elemen antarmuka pengguna. Gambar 4.105 merupakan kode *Container* pada *Welcome Page* aplikasi *mobile*.

```dart
import 'package:flutter/material.dart';

class WelcomeScreen extends StatelessWidget {
  const WelcomeScreen({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Stack(
        children: [
          // Background Image
          Container(
            decoration: BoxDecoration(
              image: DecorationImage(
                image: AssetImage('assets/images/login_bg.png'),
                fit: BoxFit.cover,
              ), // DecorationImage
            ), // BoxDecoration
          ), // Container
          // Overlay Gradient
          Container(
            decoration: BoxDecoration(
              gradient: LinearGradient(
                colors: [
                  Color(0x22337780), // Warna dengan opasitas 14%
                  Color(0xFF337780), // Warna penuh
                ],
                begin: Alignment.topCenter,
                end: Alignment.bottomCenter,
              ), // LinearGradient
            ), // BoxDecoration
          ), // Container
```

Gambar 4.105 Kode Container pada Welcome Page Aplikasi Mobile

Bagian awal dari *widget tree* dimulai dengan penggunaan *Stack*, yaitu *widget* yang memungkinkan penyusunan beberapa elemen secara tumpang tindih. Lapisan pertama dari *Stack* ini adalah *Container* yang berfungsi menampilkan gambar latar belakang (*background image*). Gambar ini diatur menggunakan properti *BoxDecoration*, dengan sumber gambar yang diambil dari direktori lokal yaitu *assets/images/login_bg.png*. Untuk memastikan gambar memenuhi seluruh layar, digunakan properti *fit: BoxFit.cover*.

Setelah gambar latar ditampilkan, ditambahkan lapisan kedua berupa *overlay gradient* menggunakan *Container* yang berbeda. *Overlay* ini berfungsi untuk memberikan efek gradasi warna yang membuat teks di atasnya lebih mudah dibaca. Efek gradasi dibentuk dengan *BoxDecoration* yang berisi *LinearGradient*, dimulai dari bagian atas layar (*Alignment.topCenter*) dan berakhir di bagian bawah (*Alignment.bottomCenter*). Gradasi ini terdiri atas dua warna, yakni warna dengan *oppacity* 14% (Color(0x22337780)) dan warna biru penuh (Color(0xFF337780)), yang secara visual memberi nuansa gelap dan elegan di seluruh tampilan *Welcome Page*. Pada Gambar 4.106 merupakan kode *widget padding* pada aplikasi *mobile*.

```
Padding(
  padding: const EdgeInsets.all(16.0),
  child: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    crossAxisAlignment: CrossAxisAlignment.center,
    children: [
      // Logo and EarMate Text (Updated size to 45 for EarMate and alignment)
      Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Image.asset(
            'assets/images/Logo_EarMate_Putih.png',
            height: 80,
          ), // Image.asset
          SizedBox(width: 15),
          Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              Text(
                'EarMate',
                style: TextStyle(
                  fontFamily: 'Poppins',
                  fontSize: 45,
                  fontWeight: FontWeight.bold,
                  color: Colors.white,
                ), // TextStyle
              ), // Text
              Text(
                'Supported By PLN Nusantara Power',
                style: TextStyle(
                  fontFamily: 'Poppins',
                  fontSize: 11,
                  fontWeight: FontWeight.w300,
                  color: Colors.white,
```

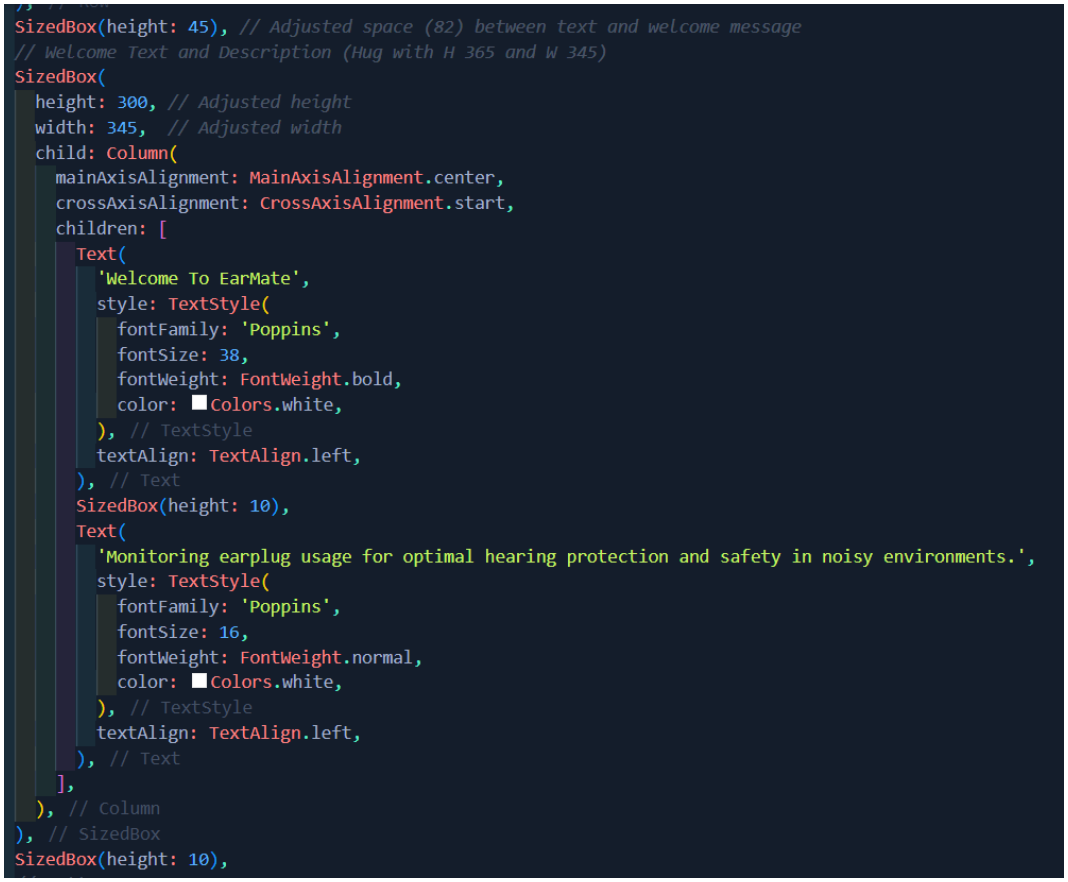
Gambar 4.106 Kode Widget Padding pada Welcome Page Aplikasi Mobile

Setelah dua lapisan awal yaitu *background image* dan *overlay gradient* selesai diatur, elemen berikutnya yang ditambahkan ke dalam halaman adalah konten utama dari *Welcome Page*. Konten ini dibungkus dalam *widget Padding* yang berfungsi memberikan jarak sebesar 16 piksel dari semua sisi, sehingga tampilan elemen di dalamnya tidak terlalu menempel ke tepi layar.

Di dalam *Padding*, digunakan *widget Column* sebagai wadah vertikal untuk menyusun berbagai elemen seperti logo, nama aplikasi, dan teks pendukung. Properti *mainAxisAlignment: MainAxisAlignment.center* dan *crossAxisAlignment: CrossAxisAlignment.center* digunakan agar seluruh isi kolom ini berada di tengah layar, baik secara vertikal maupun horizontal.

Bagian pertama dari kolom ini adalah baris yang berisi logo dan nama aplikasi. Untuk membentuk baris ini, digunakan *widget Row* dengan *alignment* tengah. Di dalamnya terdapat *Image.asset* yang memuat logo *EarMate* berwarna putih dari direktori lokal *assets/images/Logo_EarMate_Putih.png,* dengan tinggi gambar ditetapkan sebesar 80 piksel. Di sebelah logo, terdapat *SizedBox* dengan lebar 15 piksel yang berfungsi memberikan jarak antara logo dan teks.

Selanjutnya, terdapat *Column* kecil di sebelah logo untuk menampilkan dua *text label*. Teks pertama adalah "*EarMate*" yang ditampilkan menggunakan *font family Poppins*, dengan ukuran huruf cukup besar yaitu 45, dan *font weight* bold agar terlihat mencolok. Warna huruf disesuaikan dengan latar belakang yaitu putih, agar tetap terbaca dengan jelas di atas *overlay gradient*. Di bawahnya, ditambahkan teks pendukung "*Supported By PLN Nusantara Power*" dengan ukuran lebih kecil (11) dan *font weight* yang lebih ringan (w300).



```
SizedBox(height: 45), // Adjusted space (82) between text and welcome message
// Welcome Text and Description (Hug with H 365 and W 345)
SizedBox(
  height: 300, // Adjusted height
  width: 345,  // Adjusted width
  child: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    crossAxisAlignment: CrossAxisAlignment.start,
    children: [
      Text(
        'Welcome To EarMate',
        style: TextStyle(
          fontFamily: 'Poppins',
          fontSize: 38,
          fontWeight: FontWeight.bold,
          color: Colors.white,
        ), // TextStyle
        textAlign: TextAlign.left,
      ), // Text
      SizedBox(height: 10),
      Text(
        'Monitoring earplug usage for optimal hearing protection and safety in noisy environments.',
        style: TextStyle(
          fontFamily: 'Poppins',
          fontSize: 16,
          fontWeight: FontWeight.normal,
          color: Colors.white,
        ), // TextStyle
        textAlign: TextAlign.left,
      ), // Text
    ],
  ), // Column
), // SizedBox
SizedBox(height: 10),
```

Gambar 4.107 Kode Deskripsi Welcoming Page pada Aplikasi Mobile

Setelah bagian logo dan nama aplikasi, kode dilanjutkan dengan memberikan jarak vertikal menggunakan *widget SizedBox* setinggi 45 piksel. Jarak ini berfungsi untuk memisahkan elemen identitas aplikasi dari bagian sambutan agar tampilan lebih terstruktur dan tidak terlihat terlalu rapat. Selanjutnya ditambahkan sebuah *widget SizedBox* lagi dengan ukuran tetap, yaitu tinggi 300 piksel dan lebar 345 piksel.

Pada Gambar 4.107 menampilkan kode deskripsi singkat *welcoming* page pada aplikasi *mobile.* Di dalam kode ini terdapat sebuah *Column* yang digunakan untuk menyusun dua elemen teks secara vertikal, yaitu judul sambutan dan deskripsinya. Properti *mainAxisAlignment*: *MainAxisAlignment.center* membuat teks berada di tengah secara vertikal, sedangkan *crossAxisAlignment: CrossAxisAlignment.start* memastikan bahwa kedua teks ini rata kiri.

Teks pertama yang ditampilkan adalah kalimat sambutan utama "*Welcome To EarMate*". Teks ini ditata menggunakan *font family Poppins*, ukuran huruf cukup besar yaitu 38, dengan *font weight bold* agar menarik perhatian pengguna. Warna huruf putih digunakan untuk menjaga kontras terhadap latar belakang yang gelap, serta *textAlign: TextAlign.left* untuk menyelaraskan teks ke kiri agar mudah dibaca.

Setelah itu, ditambahkan *SizedBox* setinggi 10 piksel untuk memberi jarak antara judul dan deskripsi. Kemudian ditampilkan teks deskripsi singkat yaitu "*Monitoring earplug usage for optimal hearing protection and safety in noisy environments.*" Deskripsi ini menggunakan gaya huruf *Poppins* dengan ukuran sedang (16), berat huruf normal, dan tetap menggunakan warna putih serta perataan ke kiri. Tujuan dari bagian ini adalah memberikan konteks awal kepada pengguna mengenai manfaat dan tujuan aplikasi *EarMate* dengan bahasa yang ringkas dan jelas.

```
Column(
  children: [
    ElevatedButton(
      onPressed: () {
        Navigator.pushNamed(context, '/login');
      },
      style: ElevatedButton.styleFrom(
        backgroundColor: Color(0xFFF8F8F8), // Button fill color
        minimumSize: Size(double.infinity, 50),
      ),
      child: Text(
        'Login',
        style: TextStyle(
          fontFamily: 'Poppins',
          fontSize: 16,
          fontWeight: FontWeight.w600,
          color: Color(0xFF337780),
        ), // TextStyle
      ), // Text
    ), // ElevatedButton
    SizedBox(height: 16),
    OutlinedButton(
      onPressed: () {
        Navigator.pushNamed(context, '/register');
      },
      style: OutlinedButton.styleFrom(
        side: BorderSide(color: Color(0xFFF8F8F8), width: 3),
        minimumSize: Size(double.infinity, 50),
      ),
      child: Text(
        'Sign Up',
        style: TextStyle(
          fontFamily: 'Poppins',
          fontSize: 16,
          fontWeight: FontWeight.w600,
          color: Color(0xFFF8F8F8),
        ), // TextStyle
```

Gambar 4.108 Kode Column pada Welcome Page Aplikasi Mobile

Setelah menampilkan sambutan dan deskripsi aplikasi, bagian berikut dari *welcoming page* ini adalah menampilkan dua buah tombol interaktif yang memungkinkan pengguna untuk melanjutkan ke proses masuk (*login*) atau mendaftar akun baru (*sign up*). Elemen-elemen ini dibungkus dalam sebuah *widget Column* agar tersusun secara vertikal.

Pada Gambar 4.108 ditampilkan kode *column* pada *welcome page* aplikasi *mobile.* Elemen pertama dalam kolom ini adalah sebuah *widget ElevatedButton* yang berlabel *Login*. Tombol ini memiliki fungsi navigasi saat ditekan, perintah *Navigator.pushNamed(context, '/login')* akan mengarahkan pengguna menuju halaman *login*. Tombol ini diberi gaya menggunakan *styleFrom* dengan latar belakang berwarna terang (*hex* #F8F8F8) agar kontras dengan latar belakang layar yang gelap. Ukuran minimum tombol diatur menggunakan *Size(double.infinity, 50)* agar lebarnya mengikuti lebar layar secara penuh dan tingginya 50 piksel.

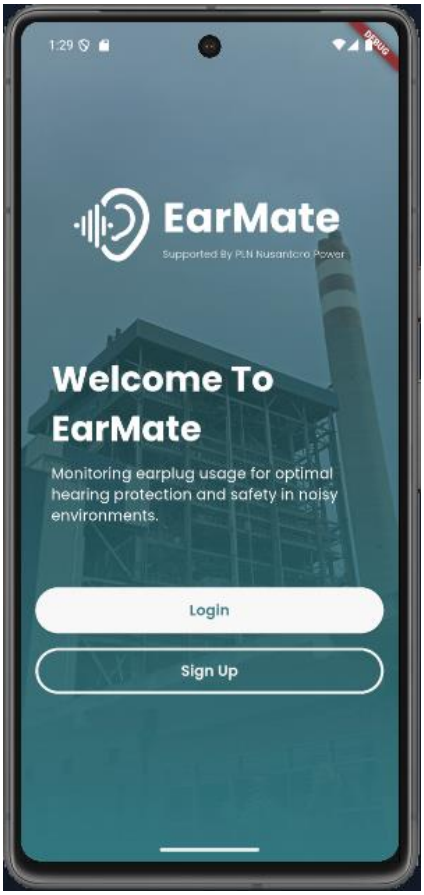
Bagian child dari tombol ini adalah teks "*Login*" yang menggunakan *font Poppins* berukuran 16 dengan *font weight* semi tebal (w600) dan warna teks biru kehijauan (*hex* #337780), yang konsisten dengan warna tema aplikasi.

Setelah tombol *Login*, ditambahkan *widget SizedBox* dengan tinggi 16 piksel sebagai jarak antar tombol untuk menciptakan tampilan yang lebih rapi dan tidak saling bertumpukan.

Tombol kedua menggunakan *OutlinedButton*, yang tampilannya berbeda karena tidak memiliki latar belakang penuh. Fungsi tombol ini adalah untuk mengarahkan pengguna ke halaman *register* menggunakan *Navigator.pushNamed(context, '/register').* Gaya tombol ini didefinisikan dengan *OutlinedButton.styleFrom,* di mana garis tepi (*border*) berwarna putih terang (#F8F8F8) dan ketebalan 3 piksel digunakan agar tetap mencolok di atas latar belakang gelap. Ukuran tombol disamakan dengan tombol sebelumnya, yaitu penuh lebar dan tinggi 50 piksel.

Teks pada tombol *Sign Up* juga menggunakan *font Poppins*, ukuran 16, *font weight* semi tebal (w600), dan warna putih agar mudah terlihat. Secara keseluruhan, kedua tombol ini tidak hanya memberikan navigasi utama bagi pengguna baru maupun lama, tetapi juga menjaga konsistensi tampilan dengan *branding* visual

aplikasi *EarMate*. Gambar 4.109 menampilkan *welcoming page* pada simulator aplikasi *mobile*.



Gambar 4.109 Welcoming Page pada Simulator Aplikasi Mobile

4.2.4.2 Pembuatan Program Login Page pada Aplikasi Mobile

Halaman *login* merupakan bagian penting dalam sebuah aplikasi karena berfungsi sebagai pintu masuk bagi pengguna untuk mengakses fitur-fitur yang tersedia. Melalui halaman ini, sistem akan melakukan proses autentikasi terhadap data yang dimasukkan, seperti *username* dan *password*. Jika data valid, maka pengguna akan diarahkan ke halaman utama atau *dashboard*. Sebaliknya, jika data tidak valid, sistem akan menampilkan pesan kesalahan.

```dart
class _LoginPageState extends State<LoginPage> {
  bool _isPasswordVisible = false;
  bool _isLoading = false; // Untuk loading state
  final TextEditingController _usernameController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();

  String? _usernameError;
  String? _passwordError;

  void _validateAndLogin() {
    setState(() {
      _usernameError = _usernameController.text.isEmpty ? 'Username cannot be empty' : null;
      _passwordError = _passwordController.text.isEmpty ? 'Password cannot be empty' : null;
    });

    if (_usernameError == null && _passwordError == null) {
      _login(); // Panggil fungsi login jika tidak ada error
    }
  }
```

Gambar 4.110 Kode Class _LoginPageState

Pada Gambar 4.110 ditampilkan kode untuk mendefinisikan *login page* menggunakan *StatefulWidget*. Pemilihan *StatefulWidget* bertujuan agar komponen pada halaman ini dapat merespons perubahan status, seperti saat pengguna mengetuk ikon visibilitas pada kolom *password*, atau saat proses *login* sedang berlangsung dan tombol ditampilkan dalam bentuk *loading spinner*. Pembuatan kelas *_LoginPageState* memungkinkan penggunaan metode *setState()* untuk memperbarui tampilan berdasarkan status aplikasi. Selanjutnya terdapat kode untuk membuat dua buah *controller* yang berfungsi menangani *input* pengguna pada kolom *username* dan *password*. *TextEditingController* memungkinkan program membaca isi dari *TextField* serta mengelolanya, seperti mengosongkan atau mengambil nilainya ketika diperlukan, misalnya saat tombol *Sign In* ditekan. Fungsi *_validateAndLogin()* yang digunakan untuk melakukan validasi awal terhadap *input* pengguna. Jika kolom *username* atau *password* kosong, maka akan ditampilkan pesan kesalahan (*error message*) secara langsung pada kolom *input*. Fungsi ini memastikan bahwa pengguna tidak bisa langsung mengirim permintaan *login* sebelum mengisi seluruh data yang dibutuhkan. Apabila validasi berhasil, maka proses selanjutnya akan dilanjutkan ke fungsi *_login()*.

```dart
Future<void> _login() async {
  setState(() {
    _isLoading = true;
  });

  try {
    // Panggil API login
    final response = await ApiService.login(_usernameController.text, _passwordController.text);
    setState(() {
      _isLoading = false;
    });

    if (response == true) {
      // Jika login berhasil, navigasi ke dashboard
      // ignore: use_build_context_synchronously
      Navigator.pushReplacementNamed(context, '/dashboard');
    } else {
      // Jika login gagal, tampilkan pesan error
      // ignore: use_build_context_synchronously
      ScaffoldMessenger.of(context).showSnackBar(SnackBar(
        content: Text('Login failed. Please check your credentials.'),
      )); // SnackBar
    }
  } catch (e) {
    setState(() {
      _isLoading = false;
    });
    // ignore: use_build_context_synchronously
    ScaffoldMessenger.of(context).showSnackBar(SnackBar(
      content: Text('Error occurred: $e'),
    )); // SnackBar
  }
}
```

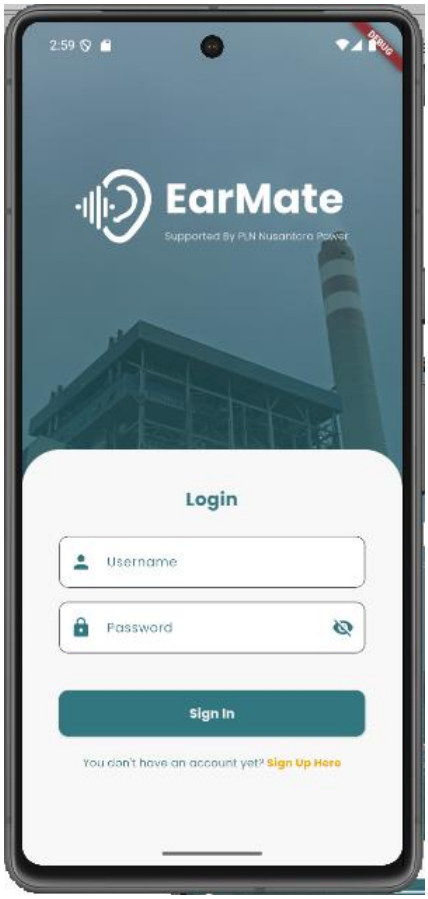
Gambar 4.111 Fungsi _login() pada Login Page Aplikasi Mobile

Pada Gambar 4.111 ditampilkan kode *fungsi _login()* yang menangani proses autentikasi pengguna. Fungsi ini memanggil layanan *API* melalui *ApiService.login()* dan mengirimkan data *username* serta *password* yang telah di-*input* oleh *user*. Jika *login* berhasil, maka *user* akan diarahkan menuju halaman *dashboard* menggunakan *Navigator.pushReplacementNamed()*. Sebaliknya, jika gagal, maka akan ditampilkan pesan kesalahan melalui *SnackBar*. Status *loading* juga diatur melalui variabel *_isLoading*, sehingga saat proses berlangsung, tampilan tombol berubah menjadi indikator proses.

```
SizedBox(
  width: double.infinity,
  child: ElevatedButton(
    onPressed: _isLoading ? null : _validateAndLogin,
    style: ElevatedButton.styleFrom(
      backgroundColor: Color(0xFF337780),
      shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.circular(10),
      ), // RoundedRectangleBorder
      padding: EdgeInsets.symmetric(vertical: 15),
    ),
```

Gambar 4.112 Kode untuk Button Sign In Pada Login Page Aplikasi Mobile

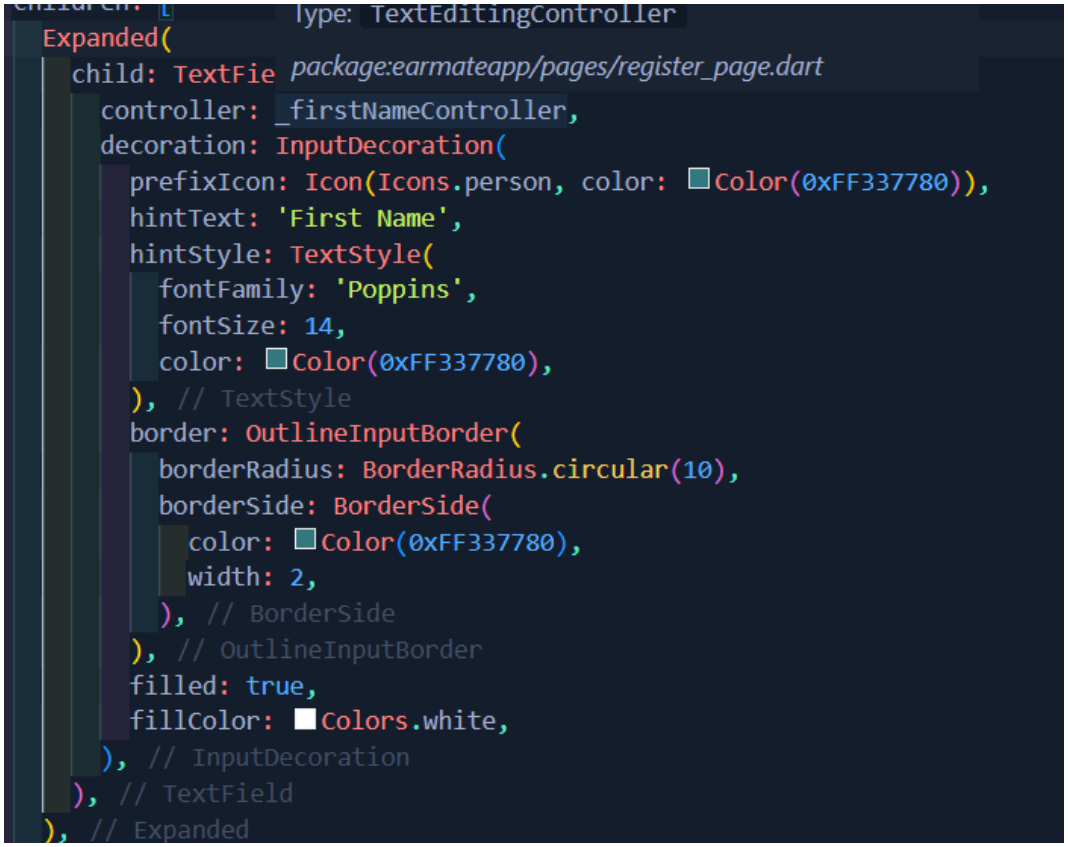
Pada Gambar 4.112 ditampilkan kode untuk tombol *Sign In* yang akan memicu proses *login* saat ditekan. Jika proses sedang berlangsung (*_isLoading = true*), maka tombol akan menampilkan *CircularProgressIndicator* sebagai indikator bahwa sistem sedang memproses permintaan. Hal ini memberikan umpan balik visual kepada pengguna agar tidak menekan tombol berulang kali dan mengetahui bahwa permintaan sedang diproses. Pada Gambar 4.113 ditampilkan visual *Login Page* aplikasi *mobile*.



Gambar 4.113 Visual Login Page Aplikasi Mobile

4.2.4.3 Pembuatan Program Register Page pada Aplikasi Mobile

Salah satu fitur penting dalam proses autentikasi pengguna adalah halaman *Register* yang memungkinkan pengguna baru untuk membuat akun. Dalam aplikasi ini, halaman *Register* dirancang dengan beberapa komponen penting yang mendukung pengalaman pengguna (*user experience*) dan memastikan data yang dimasukkan valid. Beberapa fitur utama yang tersedia dalam halaman ini antara lain adalah *text field* untuk nama depan, nama belakang, nomor induk, *username*, dan *password*, serta tombol *register* yang terhubung langsung dengan API pendaftaran pengguna. Selain itu, terdapat indikator *loading* untuk memberi tahu pengguna saat proses sedang berlangsung, serta fitur *toggle password visibility* yang mempermudah pengguna saat mengetikkan kata sandi.



Gambar 4.114 Kode Controller First Name

Pada bagian ini, digunakan sebuah *TextField* untuk menerima *input* berupa nama depan dari pengguna. *TextField* ini dikendalikan oleh *controller* bernama *_firstNameController*, yang memungkinkan aplikasi untuk mengambil isi teks dari pengguna secara langsung. Penggunaan *controller* ini penting karena nantinya nilai dari *TextField* akan dikirim ke *backend* saat proses registrasi dilakukan. Tampilan dari *TextField* diatur menggunakan *InputDecoration*, yang menambahkan *Icons.person* di sisi kiri sebagai *prefixIcon* untuk memperjelas konteks *input*. Selain itu, terdapat *hintText* bertuliskan "*First Name*" yang berfungsi sebagai teks petunjuk ketika kolom masih kosong. Gaya dari *hintText* diatur menggunakan *TextStyle* dengan jenis huruf *Poppins*, ukuran 14.

Untuk tampilan garis tepi, digunakan *OutlineInputBorder* dengan *borderRadius* sebesar 10 piksel dan garis setebal 2 piksel. Selain itu, kolom *input*

ini diberi latar belakang berwarna putih dengan mengaktifkan properti *filled* dan *fillColor*.

```
Expanded(
  child: TextField(
    controller: _lastNameController,
    decoration: InputDecoration(
      prefixIcon: Icon(Icons.person, color: Color(0xFF337780)),
      hintText: 'Last Name',
      hintStyle: TextStyle(
        fontFamily: 'Poppins',
        fontSize: 14,
        color: Color(0xFF337780),
      ), // TextStyle
      border: OutlineInputBorder(
        borderRadius: BorderRadius.circular(10),
        borderSide: BorderSide(
          color: Color(0xFF337780),
          width: 2,
        ), // BorderSide
      ), // OutlineInputBorder
      filled: true,
      fillColor: Colors.white,
    ), // InputDecoration
  ), // TextField
), // Expanded
```

Gambar 4.115 Kode Controller Last Name

Pada Gambar 4.115 menunjukkan kode *controller* untuk *Last Name.* Kode ini memiliki struktur dan logika yang hampir identik dengan *TextField* sebelumnya yang digunakan untuk memasukkan *first name*. Perbedaan utamanya terletak pada fungsi inputnya, di mana *controller* yang digunakan adalah *_lastNameController*, yang secara khusus bertugas menyimpan dan mengelola teks yang dimasukkan oleh pengguna sebagai *last name*. Selain itu, nilai dari *hintText* diubah menjadi "*Last Name*" untuk memberikan petunjuk yang sesuai dengan jenis data yang diminta.

Meskipun secara tampilan dan *decoration* tidak mengalami perubahan tetap menggunakan *Icons.person*, jenis huruf *Poppins*, dan skema warna biru kehijauan perbedaan fungsi logis ini sangat penting, karena masing-masing *controller*

nantinya akan mengirimkan data ke *backend* sesuai dengan parameter yang diminta pada proses *register*.

```
TextField(
  controller: _noIndukController,
  decoration: InputDecoration(
    prefixIcon: Icon(Icons.confirmation_number, color: Color(0xFF337780)),
    hintText: 'No Induk',
    hintStyle: TextStyle(
      fontFamily: 'Poppins',
      fontSize: 14,
      color: Color(0xFF337780),
    ), // TextStyle
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(10),
      borderSide: BorderSide(
        color: Color(0xFF337780),
        width: 2,
      ), // BorderSide
    ), // OutlineInputBorder
    filled: true,
    fillColor: Colors.white,
  ), // InputDecoration
), // TextField
SizedBox(height: 15),
```

Gambar 4.116 Kode Controller No Induk

Pada Gambar 4.116 ditampilkan kode untuk *controller* nomor induk, Kode ini merupakan *TextField* yang digunakan untuk menangani *input No Induk*, yaitu nomor identitas unik pengguna, yang biasanya digunakan untuk keperluan identifikasi dalam sistem. Perbedaan utamanya dibandingkan *TextField* sebelumnya terletak pada jenis data yang diminta. Pada bagian *controller*, digunakan *_noIndukController* yang secara khusus berfungsi menyimpan dan memantau teks yang dimasukkan sebagai *No Induk*.

Selain itu, ikon yang digunakan sebagai *prefixIcon* adalah *Icons.confirmation_number*, yang secara visual merepresentasikan angka atau kode, sehingga memberikan konteks visual yang tepat kepada pengguna. Meskipun struktur lainnya seperti *hintStyle*, *border*, dan warna tetap sama, perubahan pada *hintText* menjadi "No Induk" dan jenis ikon memperjelas bahwa *TextField* ini dimaksudkan untuk memasukkan informasi numerik atau kode unik. Data dari

controller ini nantinya juga akan dikirim bersama input lainnya ke *backend* saat proses *register* dilakukan.

```
TextField(
  controller: _usernameController,
  decoration: InputDecoration(
    prefixIcon: Icon(Icons.person, color: Color(0xFF337780)),
    hintText: 'Username',
    hintStyle: TextStyle(
      fontFamily: 'Poppins',
      fontSize: 14,
      color: Color(0xFF337780),
    ), // TextStyle
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(10),
      borderSide: BorderSide(
        color: Color(0xFF337780),
        width: 2,
      ), // BorderSide
    ), // OutlineInputBorder
    filled: true,
    fillColor: Colors.white,
  ), // InputDecoration
), // TextField
```

Gambar 4.117 Kode Controller Username

Pada Gambar 4.117 ditampilkan kode *controller* untuk *username.* Kode ini digunakan untuk menampilkan *TextField* yang berfungsi sebagai kolom *input Username* pada halaman *register*. Dari segi struktur, logika yang digunakan hampir sama dengan *TextField* sebelumnya, yaitu menetapkan *controller (_usernameController)* untuk memantau dan menyimpan *input* dari pengguna yang berupa *username*.

Perbedaan utamanya terletak pada *hintText* yang diubah menjadi "*Username*", yang memberi tahu pengguna bahwa kolom ini dimaksudkan untuk memasukkan nama pengguna unik mereka dalam sistem. Meskipun ikon yang digunakan (*Icons.person*) sama seperti pada kolom *First Name* dan *Last Name*, dalam konteks ini ikon tersebut merepresentasikan identitas pengguna secara umum, yaitu

username yang akan digunakan saat masuk ke aplikasi. Nilai dari *_usernameController* ini nantinya juga akan dikirim ke *backend* melalui fungsi *register*, sebagai bagian dari data yang diperlukan untuk membuat akun baru.

```
TextField(
  controller: _passwordController,
  obscureText: !_isPasswordVisible, // Gunakan state untuk visibilitas
  decoration: InputDecoration(
    prefixIcon: Icon(Icons.lock, color: Color(0xFF337780)),
    suffixIcon: IconButton(
      icon: Icon(
        _isPasswordVisible
            ? Icons.visibility
            : Icons.visibility_off,
        color: Color(0xFF337780),
      ), // Icon
      onPressed: () {
        setState(() {
          _isPasswordVisible = !_isPasswordVisible; // Toggle visibilitas
        });
      },
    ), // IconButton
    hintText: 'Password',
    hintStyle: TextStyle(
      fontFamily: 'Poppins',
      fontSize: 14,
      color: Color(0xFF337780),
    ), // TextStyle
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(10),
      borderSide: BorderSide(
        color: Color(0xFF337780),
        width: 2,
      ), // BorderSide
    ), // OutlineInputBorder
    filled: true,
    fillColor: Colors.white,
  ), // InputDecoration
), // TextField
```

Gambar 4.118 Kode Controller Password

Pada Gambar 4.118 ditunjukkan kode *controller* untuk *password.* Kode ini digunakan untuk menampilkan *TextField* yang berfungsi sebagai kolom *input Password* pada halaman *register*. Berbeda dari *TextField* sebelumnya, kolom ini memiliki fitur khusus berupa kemampuan untuk menyembunyikan atau

menampilkan teks sandi yang sedang diketik oleh pengguna. Hal ini dikendalikan oleh properti *obscureText* yang nilainya bergantung pada status *boolean* *_isPasswordVisible*. Jika *_isPasswordVisible* bernilai *false*, maka teks akan disembunyikan dan ditampilkan sebagai titik-titik.

Pada bagian *decoration*, ikon utama yang digunakan adalah *Icons.lock*, yang merepresentasikan keamanan atau kata sandi. Selain itu, terdapat *suffixIcon* berupa tombol yang dapat ditekan untuk mengubah visibilitas kata sandi. Ikon ini secara dinamis akan berganti antara *Icons.visibility* dan *Icons.visibility_off*, sesuai dengan status visibilitas saat ini. Fungsionalitas ini diatur dalam metode *onPressed* yang memanggil *setState()* untuk memperbarui tampilan antarmuka ketika pengguna mengklik ikon.

Teks *hintText* "*Password*" berfungsi sebagai petunjuk bahwa kolom ini dimaksudkan untuk *input* kata sandi pengguna. Sementara itu, nilai dari *_passwordController* akan digunakan untuk menangkap *input* pengguna dan dikirimkan bersama data lainnya ke *backend* saat proses registrasi.

```
onPressed: () async {
  if (_validateFields()) {
    // Tampilkan loading (opsional)
    setState(() {
      _isLoading = true;
    });

    // Panggil fungsi register dari ApiService
    bool success = await ApiService.register(
      _firstNameController.text,
      _lastNameController.text,
      _noIndukController.text,
      _usernameController.text,
      _passwordController.text,
    );

    // Sembunyikan loading setelah request selesai
    setState(() {
      _isLoading = false;
    });

    if (success) {
      // Jika registrasi berhasil, tampilkan dialog atau arahkan ke halaman lain
      _showRegistrationSuccess();
    } else {
      // Jika registrasi gagal, tampilkan pesan error
      // ignore: use_build_context_synchronously
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Registrasi gagal, coba lagi!')),
      );
    }
```

Gambar 4.119 Kode Aksi untuk Button Register

Pada Gambar 4.119 ditunjukkan kode aksi untuk *button register.* Ketika tombol ditekan, pertama-tama kode memanggil fungsi *_validateFields()*. Fungsi ini berfungsi untuk memverifikasi apakah semua *TextField* telah diisi dengan benar. Jika validasi berhasil (mengembalikan nilai *true*), maka proses dilanjutkan. Langkah pertama setelah validasi adalah menampilkan status *loading* dengan cara mengubah nilai *_isLoading* menjadi *true*, lalu memanggil *setState()* agar antarmuka diperbarui dan dapat menampilkan indikator *loading* jika tersedia.

Setelah itu, aplikasi akan memanggil fungsi register dari kelas *ApiService*, yang merupakan fungsi asinkron. Fungsi ini menerima data yang diambil dari masing-masing *controller* yaitu *_firstNameController*, *_lastNameController*,

_noIndukController, *_usernameController*, dan *_passwordController*. Kelima data tersebut merupakan informasi yang diinput pengguna dalam *form* registrasi. Fungsi register akan berkomunikasi dengan *backend* untuk mengirim data ke *database*.

Setelah proses pendaftaran selesai, nilai *loading* akan diset kembali ke *false* menggunakan *setState()* agar tampilan kembali normal. Kemudian, jika nilai success bernilai *true*, artinya registrasi berhasil, maka akan dipanggil fungsi *_showRegistrationSuccess()* yang biasanya berisi logika untuk menampilkan pesan sukses atau mengarahkan pengguna ke halaman *login*. Namun jika registrasi gagal , maka akan muncul notifikasi menggunakan *ScaffoldMessenger.of(context).showSnackBar()* untuk menampilkan pesan bahwa registrasi gagal, memberi umpan balik secara langsung kepada pengguna.

```dart
// Fungsi untuk register
static Future<bool> register(String firstname, String lastname, String noInduk, String username, String password) async {
  final response = await http.post(
    Uri.parse('$baseUrl/register.php'),
    body: {
      'firstname': firstname,
      'lastname': lastname,
      'no_induk': noInduk,
      'username': username,
      'password': password,
    },
  );

  if (response.statusCode == 200) {
    final data = jsonDecode(response.body);
    if (data['status'] == 'success') {
      return true; // Registrasi berhasil
    } else {
      print('Register Error: ${data['message']}');
      return false; // Gagal registrasi
    }
  } else {
    throw Exception('Failed to register user');
  }
}
```

Gambar 4.120 Kode Pengiriman data Registrasi ke Database

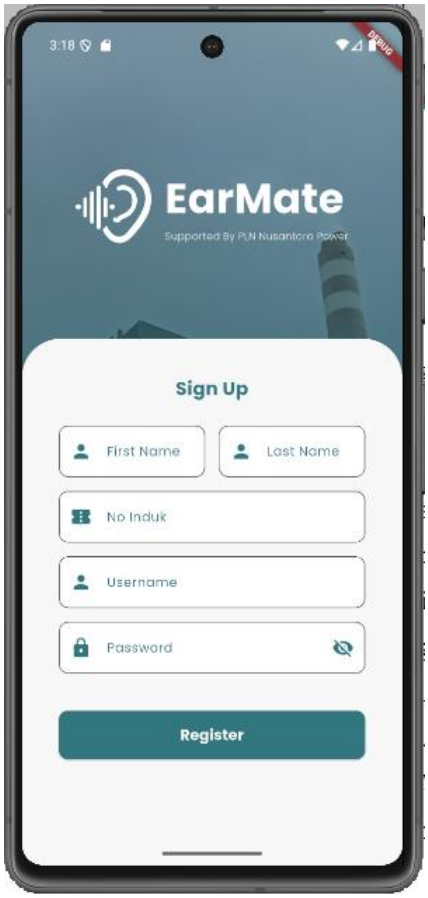
Gambar 4.120 menunjukkan kode pengiriman data Registrasi ke *Database*. Kode tersebut merupakan sebuah fungsi dalam *file api_service.dart* yang bertanggung jawab untuk mengirim data registrasi pengguna ke *server* melalui metode *HTTP POST*. Fungsi ini diberi nama register dan bersifat *static*, sehingga bisa dipanggil tanpa harus membuat *instance* dari kelas tempat fungsi ini berada.

Karena bersifat *asynchronous*, fungsi ini mengembalikan nilai berupa *Future<bool>* yang akan menyatakan apakah proses registrasi berhasil atau gagal.

Fungsi register menerima lima parameter yaitu *firstname, lastname*, noInduk, *username,* dan *password*, yang masing-masing berisi data *input* dari pengguna. Data ini dikirimkan ke *server* dengan menggunakan *http.post(),* di mana URL tujuan adalah hasil dari *'$baseUrl/register.php'*, yang berarti *server* akan menerima data melalui *endpoint register.php*. Informasi yang dikirim dalam *body* menggunakan format *form-urlencoded*, sesuai dengan standar permintaan *POST* sederhana.

Setelah permintaan terkirim, fungsi memeriksa apakah *status code* dari *respons* adalah 200, yang berarti permintaan berhasil diterima oleh *server*. Jika iya, maka isi dari *response* (dalam bentuk *JSON*) akan di-*decode* menggunakan *jsonDecode(response.body).* Kemudian dilakukan pengecekan terhadap nilai data['status']. Jika status tersebut bernilai *'success'*, maka fungsi mengembalikan nilai *true* yang menandakan bahwa registrasi berhasil. Jika tidak, maka pesan kesalahan dari server akan ditampilkan melalui *print(),* dan fungsi akan mengembalikan *false* untuk menandakan bahwa proses registrasi gagal.

Namun, jika *response* dari *server* bukan 200, maka fungsi akan langsung menghentikan proses dan melempar *exception* dengan pesan *'Failed to register user'*. Hal ini penting untuk penanganan kesalahan jika terjadi masalah jaringan atau jika *server* tidak memberikan respons yang diharapkan. Pada Gambar 4.121 ditampilkan visual *Register Page* pada Aplikasi *Mobile*.



Gambar 4.121 Tampilan Visual Register Page pada Aplikasi Mobile

4.2.4.4 Pembuatan Program Dashboard Page pada Aplikasi Mobile

Halaman dashboard pada aplikasi *mobile* merupakan fitur utama yang hanya dapat diakses oleh pengguna dengan hak akses sebagai *admin*. Halaman ini berfungsi sebagai pusat pemantauan kondisi dan aktivitas sistem dispenser earplug otomatis. Dalam halaman *dashboard*, ditampilkan berbagai informasi penting secara ringkas dan terstruktur guna mendukung pemantauan dan pengambilan keputusan terkait penggunaan serta pengisian ulang (*refill*) *earplug*.

Tampilan *dashboard* dirancang menggunakan beberapa komponen *card* dan grafik visual yang secara berkala diperbarui secara otomatis (*auto-refresh*), sehingga informasi yang ditampilkan selalu sesuai dengan kondisi terkini di lapangan. Adapun fitur-fitur utama yang terdapat pada halaman ini meliputi:

1. *Card Total Earplug Available*, yaitu komponen yang menampilkan jumlah *earplug* yang masih tersedia di dalam dispenser.
2. *Card Total Used Today*, yang berfungsi untuk menampilkan jumlah *earplug* yang telah diambil oleh pengguna pada hari berjalan.
3. Grafik Penggunaan Harian, yaitu grafik yang menampilkan data jumlah penggunaan *earplug* setiap hari dalam satu minggu terakhir.
4. Grafik Penggunaan Bulanan, yaitu grafik yang menyajikan data penggunaan *earplug* setiap bulan dalam satu tahun berjalan.
5. Grafik Penggunaan Tahunan, yang berisi data total *earplug* yang digunakan per tahun untuk beberapa tahun terakhir.

```
void _fetchHistoryData() async {
  final String formattedDate = "${selectedDate.year}-${selectedDate.month.toString()
                                .padLeft(2, '0')}-${selectedDate.day.toString().padLeft(2, '0')}";
  final url = Uri.parse('$baseUrl/get_usage_history.php?searchDate=$formattedDate');

  try {
    final response = await http.get(url);
    if (response.statusCode == 200) {
      final List<dynamic> data = json.decode(response.body);

      setState(() {
        historyData = data.map<Map<String, dynamic>>((item) => {
          'date': DateTime.parse(item['date']),
          'used_quantity': int.tryParse(item['used_quantity'].toString()) ?? 0,
        }).toList();
      });
    } else {
      print("Failed to load history: ${response.statusCode}");
    }
  } catch (e) {
    print("Error fetching history: $e");
  }
}
```

Gambar 4.122 Fungsi Pengambilan Data Penggunaan Harian

Fungsi *_fetchHistoryData()* pada Gambar 4.122 berfungsi untuk mengambil data riwayat penggunaan *earplug* berdasarkan tanggal tertentu yang dipilih oleh pengguna. Pada awal fungsi, variabel *formattedDate* dibentuk dengan format YYYY-MM-DD dari objek *selectedDate* menggunakan metode *padLeft()* untuk memastikan nilai bulan dan hari selalu terdiri dari dua digit. Format tanggal ini kemudian digunakan sebagai parameter *query string* pada alamat URL

get_usage_history.php, yang merupakan endpoint *Application Programming Interface* (*API*) berbasis PHP yang telah terhubung dengan *database* MySQL.

Selanjutnya, proses pengambilan data dilakukan menggunakan metode *http.get(url)* yang merupakan bagian dari *library* http pada Flutter. Jika *response* dari *server* memiliki kode status 200 yang menandakan permintaan berhasil, maka data yang diterima dalam format *JSON* akan di-*decode* menjadi bentuk objek *List<dynamic>*. Data tersebut kemudian diolah dan disimpan dalam variabel *historyData* dengan menggunakan *fungsisetState()*. Proses pemetaan data dilakukan melalui metode *map()* yang mengubah setiap item menjadi struktur *map* dengan dua *key*, yaitu *date* yang melewati proses *parsing* menjadi objek *DateTime*, dan *used_quantity* yang dikonversi menjadi bilangan bulat dengan fungsi *int.tryParse()*.

Apabila *response* dari server bukan 200, maka akan dicetak pesan kesalahan berupa status kode. Demikian pula, jika terjadi kesalahan saat proses pengambilan data, misalnya karena kegagalan koneksi atau *timeout*, maka akan muncul pesan kesalahan yang ditangkap melalui blok *catch*. Kode ini akan digunakan untuk menampilkan jumlah penggunaan harian *earplug* yang dipilih sesuai tanggal pada *card used today*.

```
void _fetchData() async {
final String formattedDate = "${selectedDate.year}-${selectedDate.month.toString()
                                .padLeft(2, '0')}-${selectedDate.day.toString().padLeft(2, '0')}";

final urlAvailable = Uri.parse('$baseUrl/total_earplug.php');
final urlUsedToday = Uri.parse('$baseUrl/total_used_today.php?searchDate=$formattedDate');

try {
  final responseAvailable = await http.get(urlAvailable);
  final responseUsedToday = await http.get(urlUsedToday);

  if (responseAvailable.statusCode == 200 && responseUsedToday.statusCode == 200) {
    final dataAvailable = json.decode(responseAvailable.body);
    final dataUsedToday = json.decode(responseUsedToday.body);

    setState(() {
  totalAvailable = dataAvailable['availableEarplug'];
  totalUsedToday = int.tryParse(dataUsedToday['usedToday'].toString()) ?? 0;
  });

  _fetchHistoryData();
  } else {
    print("Failed to load data: ${responseAvailable.statusCode}, ${responseUsedToday.statusCode}");
  }
} catch (e) {
  print("Error fetching data: $e");
}
}
```

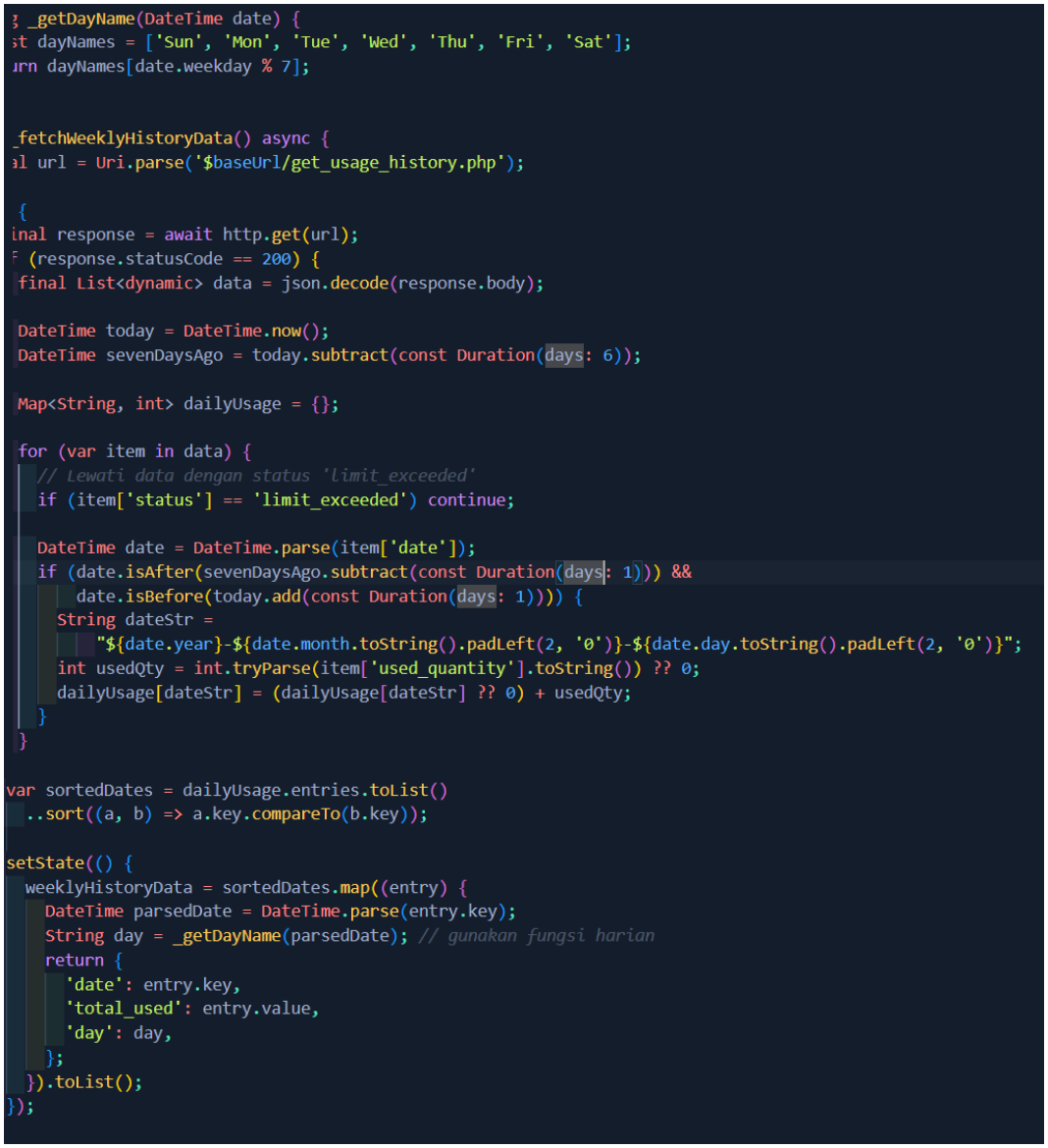
Gambar 4.123 Fungsi Pengambilan Data Jumlah Earplug Tersedia

Fungsi _fetchData() yang ditampilkan pada Gambar 4.123 merupakan bagian dari proses pengambilan data utama yang digunakan untuk menampilkan informasi pada *dashboard* aplikasi mobile. Fungsi ini akan mengambil dua jenis data, yaitu jumlah *earplug* yang masih tersedia di dalam alat (*available earplug*) dan jumlah penggunaan *earplug* pada hari tertentu (*used today*). Tanggal yang dijadikan acuan ditentukan oleh variabel *selectedDate*, kemudian diformat ke dalam bentuk YYYY-MM-DD menggunakan metode *padLeft()* agar dapat dikirim sebagai parameter *query string* pada permintaan data.

Pada bagian selanjutnya, dibuat dua buah objek Uri yang masing-masing menunjuk ke *endpoint* PHP yang berbeda, yaitu *total_earplug.php* untuk mendapatkan data jumlah *earplug* yang tersedia, dan *total_used_today.php* untuk memperoleh data jumlah penggunaan pada tanggal yang ditentukan. Proses pengambilan data dilakukan secara *asynchronous* menggunakan metode *http.get()* dari *library http* milik Flutter. Kedua *response* tersebut disimpan dalam variabel *responseAvailable* dan *responseUsedToday*.

Apabila kedua *response* tersebut memiliki kode status 200, yang berarti permintaan berhasil, maka isi dari masing-masing *response* akan di-*decode* dari format *JSON* ke dalam objek *map* dengan menggunakan fungsi json.decode(). Data yang berhasil diperoleh kemudian disimpan ke dalam variabel *totalAvailable* dan *totalUsedToday*. Proses ini dilakukan di dalam metode *setState()* agar tampilan *user interface* dapat diperbarui secara langsung sesuai dengan data terbaru. Nilai *totalUsedToday* juga diamankan dengan menggunakan *int.tryParse()* untuk memastikan bahwa data yang ditampilkan berupa bilangan bulat dan tidak menyebabkan kesalahan saat *parsing*.

Setelah data utama berhasil diambil, fungsi *_fetchHistoryData()* dipanggil untuk melanjutkan proses pengambilan data historis penggunaan *earplug* yang akan ditampilkan dalam bentuk grafik. Apabila terjadi kesalahan baik pada koneksi jaringan maupun pada struktur *response* yang diterima, maka pesan kesalahan akan ditampilkan melalui blok *catch*, atau melalui pernyataan *print()* yang menunjukkan kode status jika permintaan gagal.



```
; _getDayName(DateTime date) {
st dayNames = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'];
urn dayNames[date.weekday % 7];


_fetchWeeklyHistoryData() async {
al url = Uri.parse('$baseUrl/get_usage_history.php');

 {
inal response = await http.get(url);
f (response.statusCode == 200) {
 final List<dynamic> data = json.decode(response.body);

 DateTime today = DateTime.now();
 DateTime sevenDaysAgo = today.subtract(const Duration(days: 6));

 Map<String, int> dailyUsage = {};

 for (var item in data) {
   // Lewati data dengan status 'limit_exceeded'
   if (item['status'] == 'limit_exceeded') continue;

   DateTime date = DateTime.parse(item['date']);
   if (date.isAfter(sevenDaysAgo.subtract(const Duration(days: 1))) &&
       date.isBefore(today.add(const Duration(days: 1)))) {
     String dateStr =
         "${date.year}-${date.month.toString().padLeft(2, '0')}-${date.day.toString().padLeft(2, '0')}";
     int usedQty = int.tryParse(item['used_quantity'].toString()) ?? 0;
     dailyUsage[dateStr] = (dailyUsage[dateStr] ?? 0) + usedQty;
   }
 }

var sortedDates = dailyUsage.entries.toList()
  ..sort((a, b) => a.key.compareTo(b.key));

setState(() {
  weeklyHistoryData = sortedDates.map((entry) {
    DateTime parsedDate = DateTime.parse(entry.key);
    String day = _getDayName(parsedDate); // gunakan fungsi harian
    return {
      'date': entry.key,
      'total_used': entry.value,
      'day': day,
    };
  }).toList();
});
```

Gambar 4.124 Kode _fetchWeeklyHistoryData()

Fungsi *_fetchWeeklyHistoryData()* pada Gambar 4.124 digunakan untuk mengambil dan mengolah data historis penggunaan *earplug* dalam rentang waktu tujuh hari terakhir, yang kemudian ditampilkan dalam bentuk grafik mingguan. Proses ini diawali dengan mendefinisikan *endpoint* dari *API* melalui variabel url, yaitu mengakses skrip PHP *get_usage_history.php* yang berada di *server*.

Permintaan data dilakukan secara *asynchronous* dengan memanfaatkan fungsi *http.get()*. Apabila *response* yang diterima memiliki status kode 200, maka data yang dikembalikan diasumsikan dalam format *JSON array* dan akan di-*decode*

ke dalam bentuk *list* dari *dynamic object*. Data tersebut kemudian disaring berdasarkan waktu. Dua buah *timestamp* ditentukan sebagai batasan, yaitu *today* (hari ini) dan *sevenDaysAgo* (enam hari sebelum hari ini), sehingga total jangka waktu yang ditampilkan adalah tujuh hari.

Dalam proses iterasi terhadap *list* data, setiap entri akan diperiksa apakah status-nya adalah "*limit_exceeded*". Jika ya, maka entri tersebut dilewati karena tidak dihitung dalam statistik penggunaan. Selanjutnya, setiap tanggal akan diperiksa apakah berada dalam rentang waktu tujuh hari terakhir. Jika memenuhi syarat, maka tanggal tersebut diformat menjadi string dalam bentuk YYYY-MM-DD, dan nilai penggunaan pada tanggal tersebut disimpan atau ditambahkan ke dalam *map dailyUsage*, dengan tanggal sebagai *key* dan total jumlah *earplug* yang digunakan sebagai *value*.

Setelah proses penyaringan selesai, data dalam *dailyUsage* akan diubah menjadi *list* yang telah diurutkan berdasarkan tanggal menggunakan metode *sort()*. Data yang telah diurutkan kemudian diubah menjadi format yang dapat ditampilkan dalam grafik, yaitu berupa *list* objek yang masing-masing berisi tiga elemen: *date* (tanggal pemakaian), *total_used* (jumlah penggunaan pada tanggal tersebut), dan *day* (nama hari dalam format singkat seperti "Mon", "Tue", dan seterusnya).

Nama hari diperoleh melalui fungsi *_getDayName()* yang akan mengonversi *day index* dari *DateTime* menjadi *string* singkat. Penamaan hari ini digunakan untuk memperjelas tampilan grafik mingguan pada sumbu horizontal (*x-axis*) sehingga pengguna dapat dengan mudah mengenali pola penggunaan berdasarkan hari. Setelah seluruh data diproses, fungsi *setState()* dipanggil untuk memperbarui tampilan antarmuka agar grafik dapat menampilkan data terbaru secara otomatis.

```dart
void _fetchMonthlyHistoryData() async {
  final url = Uri.parse('$baseUrl/get_usage_history.php');

  try {
    final response = await http.get(url);
    if (response.statusCode == 200) {
      final List<dynamic> data = json.decode(response.body);

      Map<int, int> monthlyUsage = {}; // key: month (1-12), value: total used

      for (var item in data) {
        // Skip jika status-nya 'limit_exceeded'
        if (item['status'] == 'limit_exceeded') continue;

        DateTime date = DateTime.parse(item['date']);
        int usedQty = int.tryParse(item['used_quantity'].toString()) ?? 0;

        monthlyUsage[date.month] = (monthlyUsage[date.month] ?? 0) + usedQty;
      }

      var sortedMonthly = monthlyUsage.entries.toList()
        ..sort((a, b) => a.key.compareTo(b.key));

      setState(() {
        monthlyHistoryData = List.generate(12, (index) {
          int month = index + 1;
          return {
            'month': month,
            'total_used': monthlyUsage[month] ?? 0,
          };
        });
      });
```

Gambar 4.125 Kode _fetchMonthlyHistoryData()

Fungsi *_fetchMonthlyHistoryData()* pada Gambar 4.125 berfungsi untuk mengambil data historis pemakaian *earplug* dari *database* dan mengelompokkannya berdasarkan bulan, guna ditampilkan dalam bentuk grafik bulanan pada aplikasi. Proses ini dimulai dengan mendefinisikan *endpoint* berupa skrip PHP *get_usage_history.php* yang diakses melalui URL. Permintaan dilakukan secara *asynchronous* menggunakan metode *http.get()*.

Apabila *server* memberikan respons dengan status kode 200, maka data yang diperoleh diasumsikan dalam format *JSON array*, kemudian di-*decode* menjadi struktur *list*. Selanjutnya, kode melakukan iterasi terhadap setiap entri data untuk

menghitung total penggunaan *earplug* berdasarkan bulan. Namun, sebelum dilakukan pengelompokan, data yang memiliki status "*limit_exceeded*" terlebih dahulu disaring dan diabaikan, karena tidak termasuk dalam perhitungan statistik pemakaian.

Dalam proses perhitungan, setiap tanggal dalam data dikonversi menjadi objek *DateTime*. Dari objek tersebut, informasi bulan diambil melalui atribut *.month*. Nilai penggunaan (*used quantity*) kemudian ditambahkan ke dalam struktur *map monthlyUsage*, dengan *key* berupa bilangan bulat 1 hingga 12 yang merepresentasikan bulan Januari hingga Desember, dan *value* berupa akumulasi jumlah *earplug* yang digunakan dalam bulan tersebut.

Setelah semua data diproses, entri-entri dalam *map* tersebut diubah menjadi *list* dan diurutkan berdasarkan bulan menggunakan metode *sort()*. Kemudian, fungsi *setState()* dipanggil untuk memperbarui tampilan antarmuka pengguna. Dalam proses ini, data disusun ulang menjadi *list* berjumlah 12 elemen, masing-masing mewakili bulan dalam setahun. Untuk setiap bulan, dibuat struktur data berupa *map* dengan elemen *'month'* (angka 1–12) dan *'total_used'* yang berisi jumlah penggunaan *earplug* pada bulan tersebut. Jika tidak ada data penggunaan pada suatu bulan, maka akan memiliki nilai *default* 0.

```dart
void _fetchYearlyHistoryData() async {
  final url = Uri.parse('$baseUrl/get_usage_history.php');

  try {
    final response = await http.get(url);
    if (response.statusCode == 200) {
      final List<dynamic> data = json.decode(response.body);

      print("Raw API response: $data");

      Map<int, int> yearlyUsage = {}; // key: year, value: total used

      for (var item in data) {
        // Lewati data yang status-nya 'limit_exceeded'
        if (item['status'] == 'limit_exceeded') continue;

        DateTime date = DateTime.parse(item['date']);
        int usedQty = int.tryParse(item['used_quantity'].toString()) ?? 0;

        yearlyUsage[date.year] = (yearlyUsage[date.year] ?? 0) + usedQty;
      }

      var sortedYearly = yearlyUsage.entries.toList()
        ..sort((a, b) => a.key.compareTo(b.key));

      setState(() {
        yearlyHistoryData = sortedYearly.map((entry) {
          return {
            'year': entry.key,
            'total_used': entry.value,
          };
        }).toList();
      });
```

Gambar 4.126 Kode _fetchYearlyHistoryData()

Fungsi *_fetchYearlyHistoryData()* Gambar 4.126 digunakan untuk mengambil dan mengelompokkan data penggunaan *earplug* berdasarkan tahun, guna ditampilkan dalam bentuk grafik tahunan pada antarmuka *dashboard* aplikasi. Proses pengambilan data dilakukan melalui *endpoint* PHP *get_usage_history.php* dengan metode *HTTP GET* secara *asynchronous*. Data yang diterima dari *server* diasumsikan dalam format *JSON array*, kemudian didekode menjadi objek *list* di dalam aplikasi.

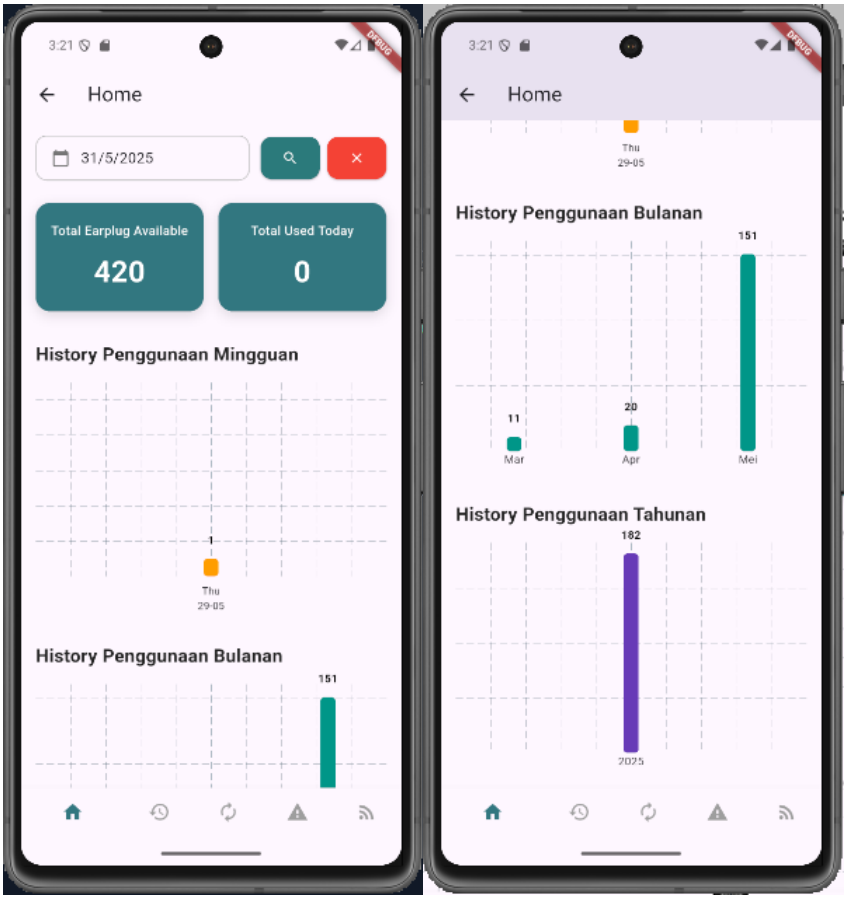
Sebelum data diproses lebih lanjut, dilakukan *filtering* terhadap data yang memiliki nilai *status* "*limit_exceeded*", agar tidak memengaruhi perhitungan

statistik tahunan. Hal ini bertujuan untuk menjaga keakuratan data yang ditampilkan pada grafik. Setiap entri dalam *list* hasil dekode diubah menjadi objek *DateTime* untuk memperoleh informasi tahun dari tanggal yang tercatat.

Selanjutnya, proses pengelompokan dilakukan menggunakan struktur *map* bernama *yearlyUsage*, dengan *key* berupa nilai tahun (tipe int), dan *value* berupa akumulasi jumlah *earplug* yang digunakan dalam tahun tersebut. Jika suatu tahun belum pernah tercatat sebelumnya, maka nilainya akan diinisialisasi dengan angka nol sebelum dilakukan penjumlahan.

Setelah proses agregasi selesai, entri-entri dalam *map yearlyUsage* diubah menjadi bentuk *list* menggunakan metode *.entries.toList()* dan diurutkan berdasarkan tahun menggunakan metode *sort()*. Kemudian, hasilnya diolah ke dalam format yang sesuai untuk kebutuhan visualisasi, yaitu *list of map* dengan masing-masing elemen berisi pasangan nilai *'year'* dan *'total_used'*.

Pembaruan antarmuka pengguna dilakukan melalui pemanggilan fungsi *setState(),* yang akan memicu *widget rebuild* untuk menampilkan grafik tahunan terbaru berdasarkan data yang telah dihimpun. Grafik ini berfungsi sebagai alat bantu analisis tren pemakaian *earplug* dalam jangka panjang. Hasil dan tampilan visual untuk *dashboard page* pada aplikasi *mobile* ditunjukkan pada Gambar 4.127.



Gambar 4.127 Tampilan Visual Dashboard pada Aplikasi Mobile

4.2.4.5 Pembuatan Program Usage History Page pada Aplikasi Mobile

Halaman *usage history* pada aplikasi mobile memiliki dua fitur utama yang memudahkan pengguna dalam mengelola dan meninjau data pengambilan *earplug*. Fitur pertama adalah tombol Unduh Laporan, yang memungkinkan pengguna untuk mengunduh seluruh data penggunaan dalam bentuk *file* untuk keperluan dokumentasi maupun pelaporan. Fitur kedua adalah tabel riwayat penggunaan yang dilengkapi dengan fungsi *pagination*, sehingga pengguna dapat melihat data dalam jumlah besar secara lebih terstruktur dan tidak membebani tampilan aplikasi. Untuk mendukung kedua fitur tersebut, aplikasi perlu melakukan proses pengambilan data dari *server* terlebih dahulu sebelum ditampilkan pada tabel atau disiapkan untuk diunduh.

```dart
Future<void> fetchUsageHistory() async {
  setState(() {
    isLoading = true;
  });

  try {
    final response = await http.get(
      Uri.parse('$baseUrl/get_usage_history.php'), // Ganti ini
    );

    if (response.statusCode == 200) {
      final List<dynamic> data = json.decode(response.body);
      allData = data.map<Map<String, String>>((item) => {
        'name': item['name'] ?? '',
        'department': item['department'] ?? '',
        'date': item['date'] ?? '',
        'time': item['time'] ?? '',
        'status': item['status'] ?? '',
      }).toList();

      setState(() {
        filteredData = allData;
        isLoading = false;
      });
    } else {
      throw Exception('Gagal mengambil data (${response.statusCode})');
    }
  } catch (e) {
    setState(() {
      isLoading = false;
    });
```

Gambar 4.128 Fungsi fetchUsageHistory() pada Usage History Page

Pada Gambar 4.128 ditunjukkan fungsi *fetchUsageHistory()* pada *Usage History Page.* Fungsi pada potongan kode di atas digunakan untuk mengambil seluruh data riwayat pengambilan *earplug* dari *server* melalui permintaan *HTTP GET* menuju *endpoint get_usage_history.php*. Permintaan ini bersifat *asynchronous*, yang berarti proses pengambilan data tidak akan memblokir jalannya program utama dan akan dijalankan di *background*. Setelah permintaan berhasil, aplikasi memverifikasi kode status dari respons. Apabila kode statusnya adalah 200, yang menandakan permintaan berhasil, maka isi dari *body* respons akan di-*dekode* dari format *JSON* ke dalam struktur data *List<dynamic>*.

Setiap entri pada *list* tersebut kemudian dipetakan menjadi struktur *map* bertipe *Map<String, String>* dengan lima atribut utama, yaitu *'name'*, *'department'*, *'date'*, *'time'*, dan *'status'*. Jika salah satu atribut tidak ditemukan dalam *JSON* yang

diterima, maka nilai kosong ('') akan digunakan sebagai nilai *default* untuk mencegah terjadinya kesalahan saat data ditampilkan di antarmuka pengguna. Hasil pemetaan ini disimpan ke dalam variabel *allData*, yang menampung seluruh data tanpa filter.

Selanjutnya, fungsi *setState()* digunakan untuk memperbarui status antarmuka aplikasi. Nilai *filteredData* akan diatur sama dengan *allData*, yang berarti seluruh data akan langsung ditampilkan ke dalam tabel sebelum dilakukan pemfilteran lebih lanjut. Selain itu, variabel *isLoading* diatur menjadi *false* untuk menunjukkan bahwa proses pemuatan data telah selesai dan komponen antarmuka dapat diperbarui sesuai kebutuhan. Proses tersebut dilakukan agar tabel dapat menampilkan data pengambilan *earplug*.

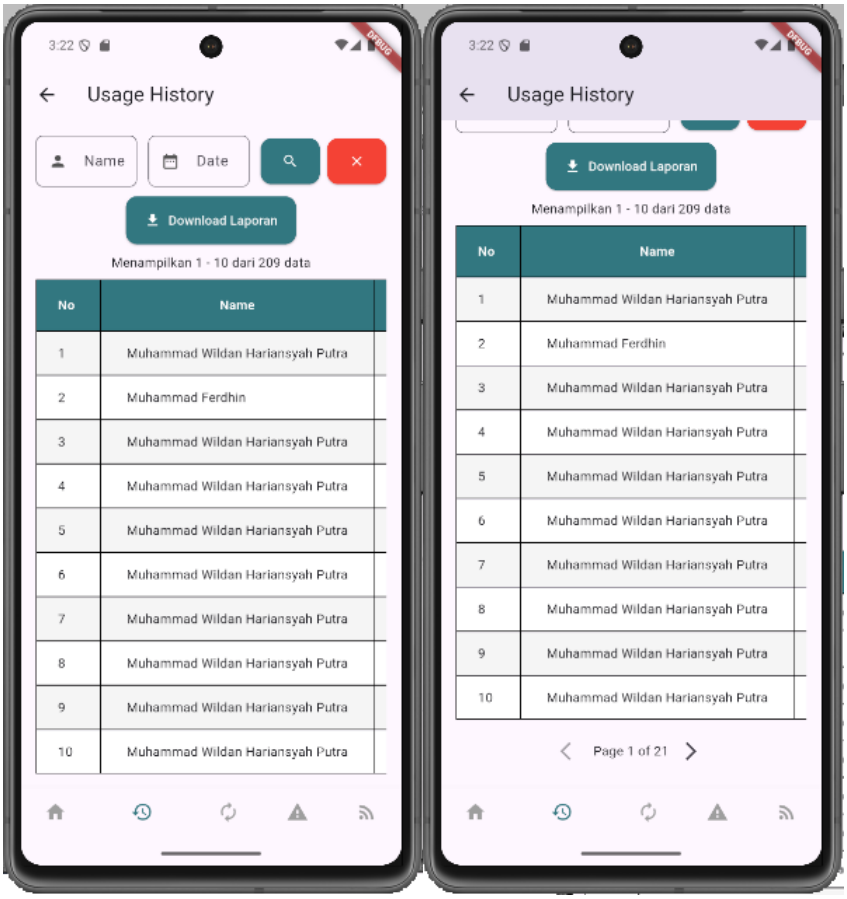
```
Future<void> _downloadExcel() async {
  final url = '$baseUrl/export_data.php';

  try {
    final dir = await getApplicationDocumentsDirectory();
    final savePath = '${dir.path}/laporan_earplug.xlsx';

    final dio = Dio();
    final response = await dio.download(url, savePath);

    if (response.statusCode == 200) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Berhasil diunduh: $savePath')),
      );

      //Membuka file setelah berhasil diunduh
        await OpenFilex.open(savePath);

    } else {
      throw Exception('Gagal mengunduh file');
    }
  } catch (e) {
    print('Error: $e');
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text('Terjadi kesalahan saat mengunduh')),
    );
  }
}
```

Gambar 4.129 Fungsi _downloadExcel()

Fungsi *_downloadExcel()* pada Gambar 4.129 merupakan fungsi *asynchronous* yang memanfaatkan paket pihak ketiga bernama Dio untuk melakukan proses pengunduhan file dari *server*. Dio merupakan *library* HTTP client untuk Flutter/Dart yang digunakan untuk melakukan permintaan jaringan (*network requests*) seperti *GET*, *POST*, *PUT*, *DELETE*, serta untuk mengunduh (*download*) atau mengunggah (*upload*) *file*. Dio termasuk salah satu *library* yang populer karena fitur-fiturnya yang lengkap dan fleksibel. Fungsi ini dimulai dengan mendefinisikan URL tujuan, yaitu *export_data.php* yang merupakan *endpoint* pada sisi *backend* untuk menghasilkan *file* Excel dari data yang tersedia. Selanjutnya, program akan mengambil direktori penyimpanan lokal aplikasi dengan memanfaatkan fungsi *getApplicationDocumentsDirectory(),* kemudian menentukan jalur lengkap tempat *file* akan disimpan, yaitu dengan nama laporan_earplug.xlsx.

Proses pengunduhan *file* dilakukan dengan memanggil *dio.download(url, savePath),* di mana url merupakan alamat *file* yang akan diunduh, dan *savePath* adalah lokasi penyimpanan lokal di perangkat. Apabila proses pengunduhan berhasil dan mengembalikan *status code* 200, maka aplikasi akan menampilkan notifikasi kepada pengguna melalui *ScaffoldMessenger*, bahwa *file* berhasil diunduh dan menunjukkan lokasi penyimpanannya.

Setelah file berhasil diunduh, aplikasi secara otomatis membuka *file* tersebut dengan memanfaatkan fungsi *OpenFilex.open(savePath).* Fungsi ini akan membuka file Excel menggunakan aplikasi pembaca file yang tersedia di perangkat pengguna, sehingga pengguna dapat langsung melihat hasil laporan tanpa harus mencarinya secara manual. Tampilan visual untuk halaman *Usage History* pada aplikasi *mobile* ditunjukkan pada Gambar 4.130.



Gambar 4.130 Tampilan Visual Usage History Page Pada Aplikasi Mobile

4.2.4.6 Pembuatan Program Refill History Page pada Aplikasi Mobile

Halaman *refill history* dirancang untuk mencatat dan menampilkan data pengisian ulang (*refill*) *earplug* yang dilakukan oleh admin. Adapun fitur utama yang tersedia dalam halaman ini meliputi kolom untuk melakukan input *refill* jumlah *earplug*, tabel yang menampilkan riwayat *refill* lengkap dengan tanggal dan jumlahnya, serta grafik yang menyajikan visualisasi data *refill* dalam bentuk bulanan dan tahunan. Fitur-fitur ini memudahkan pemantauan ketersediaan *earplug* secara historis dan akurat.

```
Future<void> _fetchRefillHistory() async {
const url = '$baseUrl/get_refill_history.php';
try {
  final response = await http.get(Uri.parse(url));
  if (response.statusCode == 302 || response.statusCode == 200) {
    final List<dynamic> jsonData = json.decode(response.body);
    setState(() {
      _data.clear();
      _data.addAll(jsonData.map((item) => {
        'refill_id': item['refill_id'],
        'date': DateFormat('dd-MM-yyyy').format(DateTime.parse(item['refill_date'])),
        'rawDate': item['refill_date'],
        'quantity': int.parse(item['earplug_count'].toString()),
      }));
    });
```

Gambar 4.131 Kode _fetchRefillHistory()

Gambar 4.131 merupakan kode _fetchRefillHistory(). Kode tersebut mengambil data dari *server* menggunakan metode *HTTP GET*. Fungsi ini dimulai dengan mendefinisikan *endpoint get_refill_history.php* sebagai sumber data yang akan diambil. Proses pengambilan data dilakukan secara *asynchronous* menggunakan pustaka *http*.

Apabila *server* merespons dengan kode status 200 (berhasil) atau 302 (pengalihan, namun tetap mengembalikan data), maka isi dari *response* akan di-*decode* dari format *JSON* menjadi objek *List* bertipe *dynamic*. Setelah data berhasil di-*decode*, program akan melakukan pembaruan terhadap *state* dengan mengosongkan data sebelumnya menggunakan *_data.clear()* dan menambahkan data baru hasil pemetaan (*mapping*) dari setiap elemen *JSON*. Setiap item yang ditambahkan memiliki beberapa atribut penting, yaitu:

1. *'refill_id'*: ID unik dari proses *refill*,
2. *'date'*: Tanggal *refill* yang telah diformat ke dalam bentuk dd-MM-yyyy menggunakan kelas *DateFormat* dari *library intl*,
3. *'rawDate'*: Tanggal asli dalam format standar ISO untuk keperluan pemrosesan grafik selanjutnya,
4. *'quantity'*: Jumlah *earplug* yang di-*refill*, diubah dari tipe data *String* menjadi *integer* agar bisa diolah secara numerik.

Seluruh proses ini dibungkus dalam blok *setState()* agar perubahan yang terjadi dapat langsung memengaruhi tampilan antarmuka aplikasi secara *real-time.* Isi tabel pada *refill history page* dipengaruhi oleh kode pada gambar 4.131 karena kode tersebut yang bertanggung jawab untuk mengambil data dari *database.*

```
void _addRefillEntry() async {
  final date = _dateController.text.trim();
  final qty = int.tryParse(_quantityController.text.trim());

  if (date.isNotEmpty && qty != null) {
    if (qty > 350) {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(content: Text('Jumlah refill tidak boleh lebih dari 350')),
      );
      return;
    }

    try {
      final response = await http.post(
        Uri.parse('$baseUrl/submit_refill.php'),
        body: {
          'refillDate': date,
          'refillQuantity': qty.toString(),
        },
      );

      if (response.statusCode == 302 || response.statusCode == 200) {
        setState(() {
          _data.insert(0, {'date': date, 'quantity': qty});
          _dateController.clear();
          _quantityController.clear();
        });

        ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(content: Text('Data berhasil ditambahkan ke database')),
        );
```

Gambar 4.132 Kode _addRefillEntry()

Gambar 4.132 ditampilkan kode untuk fungsi *_addRefillEntry()*. Fungsi *_addRefillEntry()* merupakan bagian dari sistem input data *refill* pada halaman *Refill History Page* di aplikasi mobile. Fungsi ini dirancang untuk memungkinkan pengguna—khususnya petugas atau admin—menambahkan data pengisian ulang (*refill*) *earplug* secara manual ke dalam sistem. Fungsi ini menjadi pelengkap dari fitur sebelumnya yang hanya menampilkan data *refill*, dengan memberikan kemampuan untuk memperbarui riwayat secara langsung melalui antarmuka aplikasi.

Fungsi dimulai dengan mengambil input dari dua *controller*, yaitu _dateController untuk tanggal *refill*, dan *_quantityController* untuk jumlah *earplug* yang diisi ulang. Kedua nilai ini kemudian dibersihkan dari spasi yang tidak perlu menggunakan *.trim(),* dan jumlah *earplug* diubah ke dalam bentuk bilangan bulat dengan *int.tryParse().*

Selanjutnya, dilakukan validasi untuk memastikan bahwa kedua input tidak kosong dan jumlah *refill* berada dalam batas wajar. Dalam hal ini, sistem menetapkan batas maksimal *refill* sebanyak 350 buah. Jika jumlah yang dimasukkan melebihi batas ini, maka pengguna akan diberikan notifikasi berupa *snackbar* dengan pesan peringatan, dan proses *submit* akan dihentikan dengan return.

Jika input valid, maka sistem akan mencoba mengirim data tersebut ke *server* menggunakan metode *HTTP POST* ke *endpoint submit_refill.php*. Data yang dikirim meliputi *refillDate* dan *refillQuantity*, yang masing-masing diambil dari *input* pengguna.

Apabila *server* merespons dengan kode status 200 atau 302, maka sistem menganggap bahwa proses *submit* berhasil. Sebagai bentuk umpan balik langsung, sistem akan memperbarui antarmuka aplikasi dengan menambahkan data baru ke awal daftar menggunakan *_data.insert(0, ...).* Selain itu, kolom *input* akan dikosongkan kembali menggunakan *.clear()* agar siap menerima data baru.

```
void _fetchMonthlyRefillData() async {
  final url = Uri.parse('$baseUrl/get_refill_history.php');

  try {
    final response = await http.get(url);
    if (response.statusCode == 200) {
      final List<dynamic> data = json.decode(response.body);

      Map<int, int> monthlyRefill = {}; // key: month (1-12), value: total refill

      for (var item in data) {
        DateTime date = DateTime.parse(item['refill_date']);
        int refillQty = int.tryParse(item['earplug_count'].toString()) ?? 0;

        monthlyRefill[date.month] = (monthlyRefill[date.month] ?? 0) + refillQty;
      }

      setState(() {
        monthlyRefillData = List.generate(12, (index) {
          int month = index + 1;
          return {
            'month': month,
            'total_used': monthlyRefill[month] ?? 0,
          };
        });
      });
```

Gambar 4.133 Kode _fetchMonthlyRefillData()

Kode fungsi void *_fetchMonthlyRefillData()* berfungsi untuk menampilkan data grafik pengisian ulang (*refill*) *earplug* dalam skala bulanan pada halaman *Refill History Page* di aplikasi mobile. Fitur grafik ini memberikan visualisasi yang jelas mengenai tren jumlah *refill* yang dilakukan selama satu tahun, sehingga memudahkan pengguna dalam melakukan analisis penggunaan *earplug* dari waktu ke waktu.

Proses dalam fungsi ini dimulai dengan pemanggilan *endpoint get_refill_history.php* melalui metode *HTTP GET*. Jika *server* memberikan respons dengan kode status 200 (berhasil), maka data hasil respons di-*decode* dari format *JSON* menjadi tipe data *List<dynamic>*. Selanjutnya, dilakukan proses agregasi data berdasarkan bulan. Untuk setiap item dalam data, tanggal *refill* diambil dan diubah ke dalam format *DateTime* menggunakan *DateTime.parse()*. Jumlah *earplug*

yang di-*refill* kemudian diekstraksi dan dikonversi menjadi integer menggunakan *int.tryParse()*.

Data kemudian dikelompokkan berdasarkan bulan melalui struktur *Map<int, int>* dengan bulan sebagai *key* (angka 1–12) dan total *refill* sebagai *value*. Jika dalam satu bulan sudah ada nilai sebelumnya, maka jumlah *refill* akan ditambahkan ke nilai tersebut. Setelah proses agregasi selesai, bagian *setState()* akan memicu pembaruan antarmuka aplikasi dengan memuat data grafik yang telah diproses

```
void _fetchYearlyRefillData() async {
  final url = Uri.parse('$baseUrl/get_refill_history.php');

  try {
    final response = await http.get(url);
    if (response.statusCode == 200) {
      final List<dynamic> data = json.decode(response.body);

      Map<int, int> yearlyRefill = {}; // key: year, value: total refill

      for (var item in data) {
        DateTime date = DateTime.parse(item['refill_date']);
        int refillQty = int.tryParse(item['earplug_count'].toString()) ?? 0;

        yearlyRefill[date.year] = (yearlyRefill[date.year] ?? 0) + refillQty;
      }

      var sortedYearly = yearlyRefill.entries.toList()
        ..sort((a, b) => a.key.compareTo(b.key));

      setState(() {
        yearlyRefillData = sortedYearly.map((entry) {
          return {
            'year': entry.key,
            'total_used': entry.value,
          };
        }).toList();
      });

      print("Yearly Refill Data: $yearlyRefillData");
    } else {
      print("Failed to load yearly refill data: ${response.statusCode}");
    }
  } catch (e) {
    print("Error fetching yearly refill data: $e");
  }
```

Gambar 4.134 Kode _fetchYearlyRefillData()

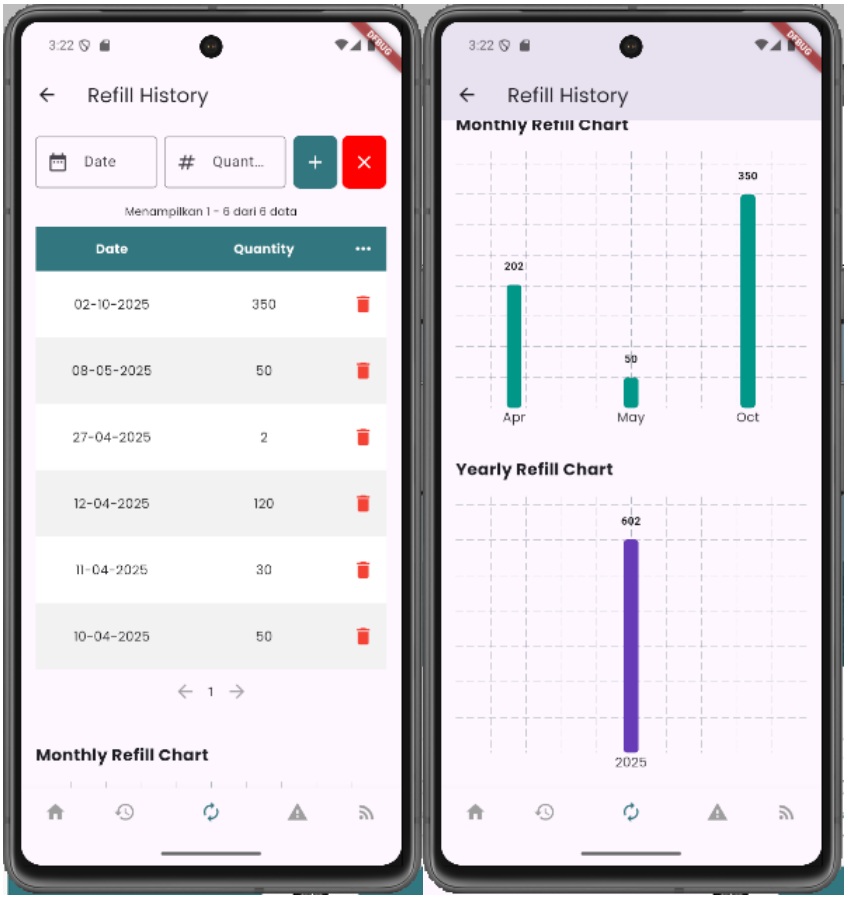
Fungsi void *_fetchYearlyRefillData()* pada Gambar 4.134 berfungsi untuk mengambil dan mengolah data *refill earplug* dalam skala tahunan. Fitur ini merupakan bagian dari tampilan grafik tahunan pada halaman *Refill History Page* dalam aplikasi *mobile*, yang bertujuan untuk memberikan gambaran mengenai total jumlah pengisian ulang *earplug* dari tahun ke tahun secara ringkas dan terstruktur.

Proses pertama dari fungsi ini adalah mengakses data dari *server* melalui *endpoint get_refill_history.php* menggunakan metode *HTTP GET*. Apabila *server* merespons dengan status kode 200, maka data yang diperoleh akan di-*decode* dari format *JSON* ke dalam bentuk *List<dynamic>*.

Selanjutnya, dilakukan proses iterasi terhadap setiap item data untuk mengambil informasi tanggal *refill* yang kemudian dikonversi menjadi objek *DateTime*. Dari objek tanggal ini, nilai tahun (*date.year*) akan diambil sebagai *key*, dan jumlah *refill* yang dilakukan (*earplug_count*) akan dijadikan *value*. Jika dalam tahun yang sama sudah terdapat nilai sebelumnya, maka jumlah *refill* akan dijumlahkan dengan nilai sebelumnya menggunakan operasi penambahan akumulatif.

Data yang telah dikelompokkan berdasarkan tahun ini kemudian diubah ke dalam bentuk *List<Map<String, dynamic>>* menggunakan fungsi *map()*, yang berisi pasangan *key-value* berupa *'year'* dan *'total_used'*. Sebelum proses *mapping*, data terlebih dahulu diurutkan secara menaik berdasarkan tahun menggunakan metode *sort()* agar hasil visualisasi nantinya tersaji secara kronologis.

Terakhir, *setState()* digunakan untuk memperbarui *state* aplikasi agar grafik tahunan menampilkan data yang telah diproses. Proses ini memungkinkan pengguna untuk melihat perkembangan konsumsi *earplug* dari tahun ke tahun secara menyeluruh. Tampilan visual *Refill History Page* dapat dilihat pada Gambar 4.135.



Gambar 4.135 Tampilan Visual Refill History Page Pada Aplikasi Mobile

4.2.4.7 Pembuatan Program Emergency Access Page pada Aplikasi Mobile

Halaman *Emergency Access* merupakan salah satu bagian penting dalam aplikasi mobile yang dikembangkan untuk mendukung sistem pengambilan *earplug* menggunakan kartu RFID. Fitur ini diperuntukkan bagi pengguna atau karyawan yang memiliki kebutuhan mendesak untuk mengambil *earplug* di luar batas normal penggunaan. Secara umum, sistem hanya memperbolehkan pengguna melakukan maksimal dua kali *tap* kartu RFID per hari untuk mengambil *earplug*. Namun, melalui halaman ini, *admin* diberikan keleluasaan untuk menyesuaikan batas jumlah *tap* sesuai dengan kebutuhan masing-masing karyawan.

Tampilan utama halaman ini berupa sebuah *data table* yang memuat informasi nama pengguna dan batas *tap* harian mereka. Setiap baris pada tabel dilengkapi dengan tombol *edit*, yang memungkinkan *admin* untuk memperbarui

jumlah batasan *tap* tanpa harus mengakses sistem secara manual melalui *backend*. Salah satu bagian penting dari implementasi fitur ini adalah penggunaan *widget IconButton* yang digunakan untuk memicu pengisian ulang nilai pada *form input*, agar bisa diedit oleh *admin.*

```
IconButton(
  icon: const Icon(Icons.edit, color: Color(0xFF337780)),
  onPressed: () {
    setState(() {
      _nameController.text = entry['nama'] ?? '';
      _limitController.text = entry['tap_limit'].toString();
    });
  },
), // IconButton
```

Gambar 4.136 Widget IconButton pada Emergency Access Page

Kode di atas menampilkan ikon *edit* berwarna biru tua yang digunakan untuk mengaktifkan mode pengeditan data pada salah satu entri di tabel. Ketika tombol ini ditekan (*onPressed*), fungsi *setState()* akan dipanggil untuk memperbarui nilai dalam dua *controller* yaitu *_nameController* dan *_limitController*.

1. *_nameController* digunakan untuk menampilkan nama pengguna yang akan diedit. Nilainya diisi dari entri data dengan *key* 'nama', dan apabila nilainya *null*, maka akan diisi dengan string kosong.
2. *_limitController* digunakan untuk menampilkan jumlah batas *tap* yang sudah disimpan sebelumnya di *database*. Nilainya diambil dari entri dengan *key* 'tap_limit', lalu dikonversi menjadi bentuk *string* agar dapat ditampilkan dalam *TextField*.

Dengan mekanisme ini, data pengguna yang ingin diubah akan secara otomatis muncul di dalam *form input*, sehingga *admin* tidak perlu mengisi ulang secara manual. Seluruh proses ini bertujuan untuk memberikan kemudahan dan kecepatan dalam proses pengelolaan akses darurat terhadap pengambilan *earplug*.

```dart
Future<void> _fetchData() async {
  setState(() {
    _isLoading = true;
  });

  try {
    final response = await http.get(Uri.parse(apiUrl));
    if (response.statusCode == 200) {
      final jsonData = json.decode(response.body);
      if (jsonData['success'] == true) {
        setState(() {
          _data = List<Map<String, dynamic>>.from(jsonData['data']);
          _currentPage = 0; // Reset halaman saat data di-refresh
        });
      } else {
        print("Error: ${jsonData['message']}");
      }
    } else {
      print("Failed to fetch data: ${response.statusCode}");
    }
  } catch (e) {
    print("Exception occurred: $e");
  } finally {
    setState(() {
      _isLoading = false;
    });
  }
}
```

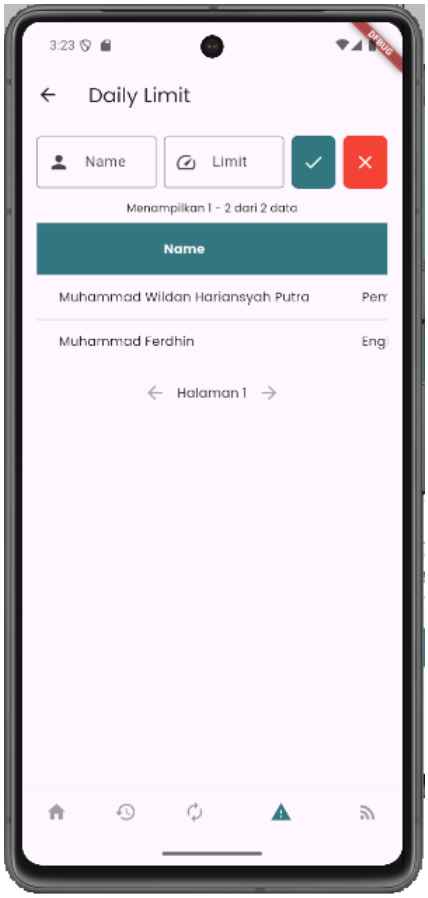
Gambar 4.137 Kode _fetchData() pada Emergency Access Page

Pada Gambar 4.137 ditunjukkan kode *_fetchData()* pada *emergency access page.* Fungsi *_fetchData()* pada aplikasi *mobile* berperan penting dalam menampilkan informasi terkini terkait data pengguna yang memiliki akses *emergency* untuk pengambilan *earplug*. Fungsi ini ditulis dalam bentuk *asynchronous* agar proses pengambilan data dari *server* tidak mengganggu *user interface* selama proses berlangsung. Pada awal eksekusi fungsi, sistem terlebih dahulu mengubah status *_isLoading* menjadi *true* melalui pemanggilan *setState()*. Hal ini berguna untuk memunculkan indikator pemuatan (*loading indicator*) yang memberi tahu pengguna bahwa data sedang diambil dari *server*.

Selanjutnya, sistem mengirim permintaan menggunakan metode *HTTP GET* ke alamat *Application Programming Interface (API)* yang telah didefinisikan dalam variabel *apiUrl*. Setelah mendapatkan respons dari *server*, dilakukan pengecekan terhadap nilai *status code* dari respons tersebut. Jika bernilai 200, artinya permintaan berhasil diproses. Kemudian data yang diterima akan di-*decode* dari format *JSON* menggunakan fungsi *json.decode()* untuk diubah menjadi struktur data yang dapat dibaca oleh aplikasi.

Setelah proses *decode* berhasil, sistem memeriksa nilai dari atribut *'success'*. Jika bernilai *true*, maka data yang tersimpan di dalam atribut *'data'* akan dimasukkan ke dalam variabel *_data* dengan format *List<Map<String, dynamic>>*. Selain itu, nilai *_currentPage* diatur ulang ke nol agar tampilan data dimulai dari halaman pertama setiap kali data diperbarui.

Pada bagian akhir fungsi, terdapat blok *finally* yang akan dijalankan terlepas dari hasil proses sebelumnya. Di dalam blok ini, nilai *_isLoading* dikembalikan menjadi *false* agar indikator pemuatan dapat disembunyikan dan pengguna dapat kembali berinteraksi dengan halaman aplikasi secara normal. Pada Gambar 4.138 ditunjukkan tampilan visual *emergency access page*. Pada halaman ini disediakan sebuah tabel yang berfungsi untuk menampilkan data pengguna. Informasi yang ditampilkan pada tabel tersebut diperoleh melalui pemanggilan fungsi *_fetchData()*.



Gambar 4.138 Tampilan Visual Emergency Access Page Pada Aplikasi Mobile

4.2.4.8 Pembuatan Program Register RFID Page pada Aplikasi Mobile

```
Future<void> sendDataToServer() async {
  final url = Uri.parse('$baseUrl/register_rfid.php');

  try {
    final response = await http.post(
      url,
      body: {
        'rfidName': _nameController.text,
        'rfidDivision': _divisionController.text,
        'rfidNoInduk': _idController.text,
        'rfidTag': _rfidController.text,
      },
    );

    if (response.statusCode == 200 || response.statusCode == 302) {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(content: Text('RFID berhasil dikirim ke server!')),
      );
    } else {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Gagal: ${response.reasonPhrase}')),
      );
    }
  } catch (e) {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Error: $e')),
    );
  }
}
```

Gambar 4.139 Kode sendDataToServer()

Gambar 4.139 menunjukkan kode *sendDataToServer()*. Fungsi *sendDataToServer()* merupakan bagian dari proses *register* kartu RFID yang terdapat pada halaman *Emergency Access* di aplikasi *mobile*. Pada halaman ini, admin memiliki kewenangan untuk menambahkan data pengguna yang akan diberi akses pengambilan *earplug* dengan cara melakukan *tap* kartu RFID. Data yang dimasukkan oleh *admin* mencakup nama pengguna, divisi, nomor induk karyawan, dan *tag* RFID. Semua informasi tersebut nantinya akan dikirim ke *server* agar tersimpan dan dapat digunakan dalam proses identifikasi saat *tap* RFID dilakukan di alat.

Kode fungsi ini diawali dengan mendefinisikan variabel url yang berisi alamat *endpoint* API *register_rfid.php*. *Endpoint* tersebut merupakan target pengiriman data ke *server*. Pengiriman data dilakukan dengan metode *HTTP POST* melalui objek *http.post*, di mana *body* permintaan berisi data yang diambil dari masing-masing *controller input*, seperti *_nameController*, *_divisionController*, *_idController*, dan *_rfidController*.

Jika server merespons dengan *status code* 200 (berhasil) atau 302 (redirect yang masih dianggap berhasil dalam konteks ini), maka aplikasi akan menampilkan *SnackBar* berisi pesan bahwa data RFID berhasil dikirim ke *server*. Namun, apabila status yang diterima bukan termasuk dua kode tersebut, maka akan ditampilkan *SnackBar* lain yang menyatakan kegagalan, disertai alasan yang diberikan oleh *server* melalui properti *reasonPhrase*. Selain itu, apabila terjadi kesalahan seperti kegagalan koneksi atau pengecualian (*exception*) saat proses berlangsung, pesan gagal akan ditampilkan menggunakan *SnackBar* untuk memberi tahu pengguna bahwa terjadi kesalahan saat pengiriman data.

```
void _register() async {
  if (_formKey.currentState!.validate()) {
    final confirm = await showDialog<bool>(
      context: context,
      builder: (context) => AlertDialog(
        title: const Text('Konfirmasi'),
        content: const Text('Apakah Anda yakin ingin mendaftarkan data ini?'),
        actions: [
          TextButton(
            onPressed: () => Navigator.pop(context, false),
            child: const Text('Batal'),
          ), // TextButton
          TextButton(
            onPressed: () => Navigator.pop(context, true),
            child: const Text('Ya'),
          ), // TextButton
        ],
      ), // AlertDialog
    );

  if (confirm == true) {
    await sendDataToServer();
    _clearFieldsWithoutDialog();
  }
  } else {
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text('Harap isi semua data!')),
    );
  }
}
```

Gambar 4.140 Kode _register()

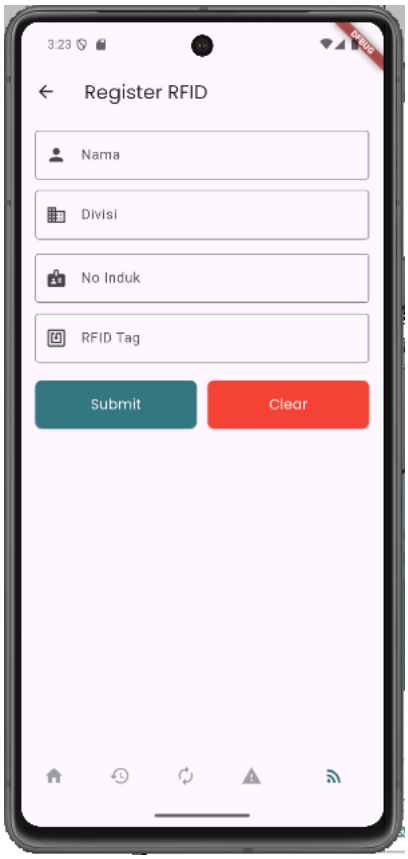
Gambar 4.140 menunjukkan kode *_register()*. Fungsi *_register()* merupakan bagian dari proses pendaftaran data pengguna RFID pada halaman *Emergency Access* aplikasi *mobile*. Fungsi ini bertanggung jawab untuk memvalidasi formulir *input*, meminta konfirmasi dari pengguna sebelum data dikirim, dan kemudian memanggil fungsi pengiriman data ke *server* apabila semua langkah berjalan dengan benar.

Proses dimulai dengan pemeriksaan validitas *form* menggunakan *_formKey.currentState!.validate()*. Pemeriksaan ini memastikan bahwa seluruh isian formulir telah diisi dengan benar dan sesuai ketentuan, seperti tidak boleh kosong. Apabila validasi berhasil, maka sistem akan menampilkan kotak dialog

konfirmasi menggunakan komponen *AlertDialog*. Dialog ini berisi pertanyaan "Apakah Anda yakin ingin mendaftarkan data ini?" dan dua tombol, yaitu Batal dan Ya. Tombol Batal akan menutup dialog dan mengembalikan nilai *false*, sedangkan tombol Ya mengembalikan *true*.

Jika pengguna memilih "Ya", maka nilai *confirm* akan menjadi *true*, sehingga fungsi *sendDataToServer()* akan dijalankan untuk mengirim data ke *server*. Setelah pengiriman berhasil, fungsi *_clearFieldsWithoutDialog()* dipanggil untuk mengosongkan semua isian *input* tanpa menampilkan dialog tambahan.

Namun, apabila formulir belum diisi dengan lengkap atau tidak valid, maka akan muncul pesan berupa *SnackBar* dengan teks "Harap isi semua data!". Fitur ini bertujuan untuk memberikan umpan balik langsung kepada pengguna agar melengkapi data terlebih dahulu sebelum melakukan pendaftaran. Pada Gambar 4.141 menunjukkan tampilan visual *register RFID*.



Gambar 4.141 Tampilan Visual Register RFID Page Pada Aplikasi Mobile