By Aamir 18 Apr 2024
Edited: 30 Apr 2024

Ahln Project

Introduction:

Our vision is to create a comprehensive and efficient package delivery system that adapts to the demands of today's dynamic market landscape thus redefining convenience and elevating the customer experience.

- **Anywhere, Anytime Delivery**: Users need not remain tethered to their homes. Whether at work, on vacation, or simply on the move, they can receive packages securely and remotely via our innovative **Ahln box** solution.

Our project seeks to transform the package delivery industry by integrating modern technologies. Through strategic combination of **PostgreSQL**, **MQTT messaging**, and **mobile applications**, we aim to achieve the following:

1. **Streamlined Operations**: By optimizing processes and minimizing bottlenecks, we enhance efficiency across the entire delivery lifecycle.
2. **Enhanced Security**: Robust authentication protocols and real-time tracking mechanisms ensure secure package handling.
3. **Improved Customer Satisfaction**: Transparent communication, accurate delivery estimates, and seamless interactions empower our users.

By embracing modern software and hardware innovations, we strive to enhance the overall package delivery experience, optimize resource utilization, and foster greater customer confidence.

At the heart of the system lies a full-bodied database management system powered by PostgreSQL, which stores critical information about customers, delivery boxes, vendors, operations, and delivery transactions. This database serves as the backbone of the entire ecosystem, enabling efficient data retrieval, storage, and analysis.

Moreover, the use of MQTT messaging protocol enables real-time communication between the delivery boxes and the customer's mobile application. This ensures seamless authentication and authorization processes, enhancing security and reliability throughout the delivery journey.

With the proliferation of mobile devices, our project places a strong emphasis on developing intuitive and user-friendly mobile applications for both customers and delivery personnel. These

applications provide a seamless user experience, allowing users to easily track their packages, communicate with delivery personnel, and receive timely updates on their delivery status.

Overall, our project aims to transform the package delivery industry by leveraging modern technologies to streamline operations. By combining the power of PostgreSQL, MQTT messaging, and mobile applications, we aspire to create a comprehensive and efficient package delivery system that meets the demands of today's dynamic market landscape.

Components of the Project:

1. Delivery Box:
    - The delivery box serves as the physical interface between the delivery personnel and the customer.
    - It is equipped with sensors, authentication mechanisms, and communication modules to facilitate secure package delivery.
    - It is also equipped with QR Code and Bar Code scanners to scan packages
    - Each delivery box is assigned a unique identifier and is associated with a specific customer and delivery address.
2. Firmware:
    - The firmware of the delivery box is responsible for controlling its operation and facilitating communication with the central server.
    - It manages authentication processes, monitors sensor data, and handles package delivery transactions.
    - The firmware ensures the secure and efficient operation of the delivery box throughout the delivery process.
3. DBMS (Database Management System):
    - The DBMS stores and manages all data related to customers, delivery boxes, vendors, operations, and delivery transactions.
    - It provides a centralized repository for storing and retrieving critical information, enabling seamless coordination between different components of the system.
    - PostgreSQL is used as the backend database system, offering reliability, scalability, and advanced data management capabilities.
4. MQTT (Message Queuing Telemetry Transport):
    - MQTT is a lightweight messaging protocol designed for use in constrained environments and low-bandwidth, high-latency, or unreliable networks.
    - It facilitates real-time communication between the delivery boxes, mobile applications, and the central server.
    - MQTT ensures efficient message delivery, minimal overhead, and reliable communication, making it ideal for use in IoT (Internet of Things) applications like the package delivery system.
5. Node.js App:
    - The Node.js application serves as the backend server that orchestrates the entire package delivery system.

- It handles incoming requests from delivery boxes, mobile applications, and other components of the system.
- The Node.js app interfaces with the DBMS to store and retrieve data, processes MQTT messages, and coordinates package delivery operations.

6. Mobile Apps:
    - Mobile applications are developed for both customers and delivery personnel to interact with the package delivery system.
    - Customers can use the mobile app to track their packages, communicate with delivery personnel, and receive notifications about their delivery status.
    - Delivery personnel can use the mobile app to view delivery assignments, scan package barcodes, and update delivery statuses in real-time.

Overall, these components work together to create a comprehensive package delivery system that offers real time updates to both customer and delivery personnel.

Node.js App:

The Node.js application serves as the backbone of the package delivery system, providing a centralized platform for managing and coordinating all operations. Here's a breakdown of its key components and functionalities:

1. Server Setup:
    - The Node.js app sets up an HTTP server using frameworks like Express.js to handle incoming requests from various sources, including delivery boxes, mobile apps, and other components.
    - It listens for requests on specified endpoints and routes them to appropriate handlers for processing.
2. Middleware:
    - Middleware functions are used to intercept and process incoming requests before they reach the route handlers.
    - Middleware can perform tasks such as request parsing, authentication, error handling, and logging, enhancing the reliability and security of the application.
3. Database Interaction:
    - The Node.js app interfaces with the PostgreSQL database to store and retrieve data related to customers, delivery boxes, vendors, operations, and delivery transactions.
    - It uses database drivers like pg-promise or Sequelize to execute SQL queries, transactions, and operations on the database.
    - The app implements CRUD (Create, Read, Update, Delete) functionalities to manage the database effectively.
4. Business Logic:
    - The core business logic of the package delivery system is implemented within the Node.js application.

- It includes functionalities for package tracking, delivery assignment, authentication, notification management, and transaction processing.
- The business logic ensures the smooth operation of the system and enforces business rules and constraints.

5. MQTT Integration:
    - The Node.js app integrates with MQTT brokers to facilitate real-time communication between delivery boxes, mobile apps, and the central server.
    - It subscribes to MQTT topics to receive messages from delivery boxes and publishes messages to trigger actions or notifications.
    - MQTT integration enables seamless coordination and synchronization between different components of the system.

Overall, the Node.js app serves as a versatile and scalable platform for building and managing the package delivery system, offering flexibility, performance, and reliability to meet the demands of modern logistics operations.

DBMS (Database Management System):

The Database Management System (DBMS) forms the foundation of the package delivery system, providing a centralized repository for storing, managing, and retrieving data. Here's an overview of its role and functionalities:

1. Data Storage:
    - The DBMS stores structured data related to customers, delivery boxes, vendors, operations, and delivery transactions in relational tables.
    - It ensures data integrity, consistency, and durability, following ACID (Atomicity, Consistency, Isolation, Durability) properties to maintain the reliability of the stored information.
2. Schema Design:
    - The schema design defines the structure of the database tables, including the attributes, data types, constraints, and relationships between entities.
    - It ensures efficient data organization, normalization, and indexing to optimize query performance and storage efficiency.
3. Query Processing:
    - The DBMS processes SQL (Structured Query Language) queries and commands to retrieve, insert, update, and delete data from the database.
    - It employs query optimization techniques, query planners, and execution engines to optimize query execution and minimize response times.
4. Transaction Management:
    - The DBMS manages database transactions, ensuring atomicity, consistency, isolation, and durability of operations performed within transaction boundaries.
    - It supports concurrent access and transaction isolation levels to maintain data consistency and integrity in multi-user environments.

5. Security and Access Control:
    - The DBMS implements security mechanisms such as authentication, authorization, and encryption to protect sensitive data from unauthorized access and manipulation.
    - It defines access control policies and user roles to restrict access to database resources based on user privileges and permissions.

Overall, the DBMS serves as a reliable and scalable data storage solution for the package delivery system, offering robust features, performance, and security to meet the demands of modern logistics operations.

Interaction Among Various APPs

Outline of the sequence and process flow in the package delivery system, highlighting the interactions among MQTT, DBMS, Node.js app, owner mobile app, and delivery guy mobile app:

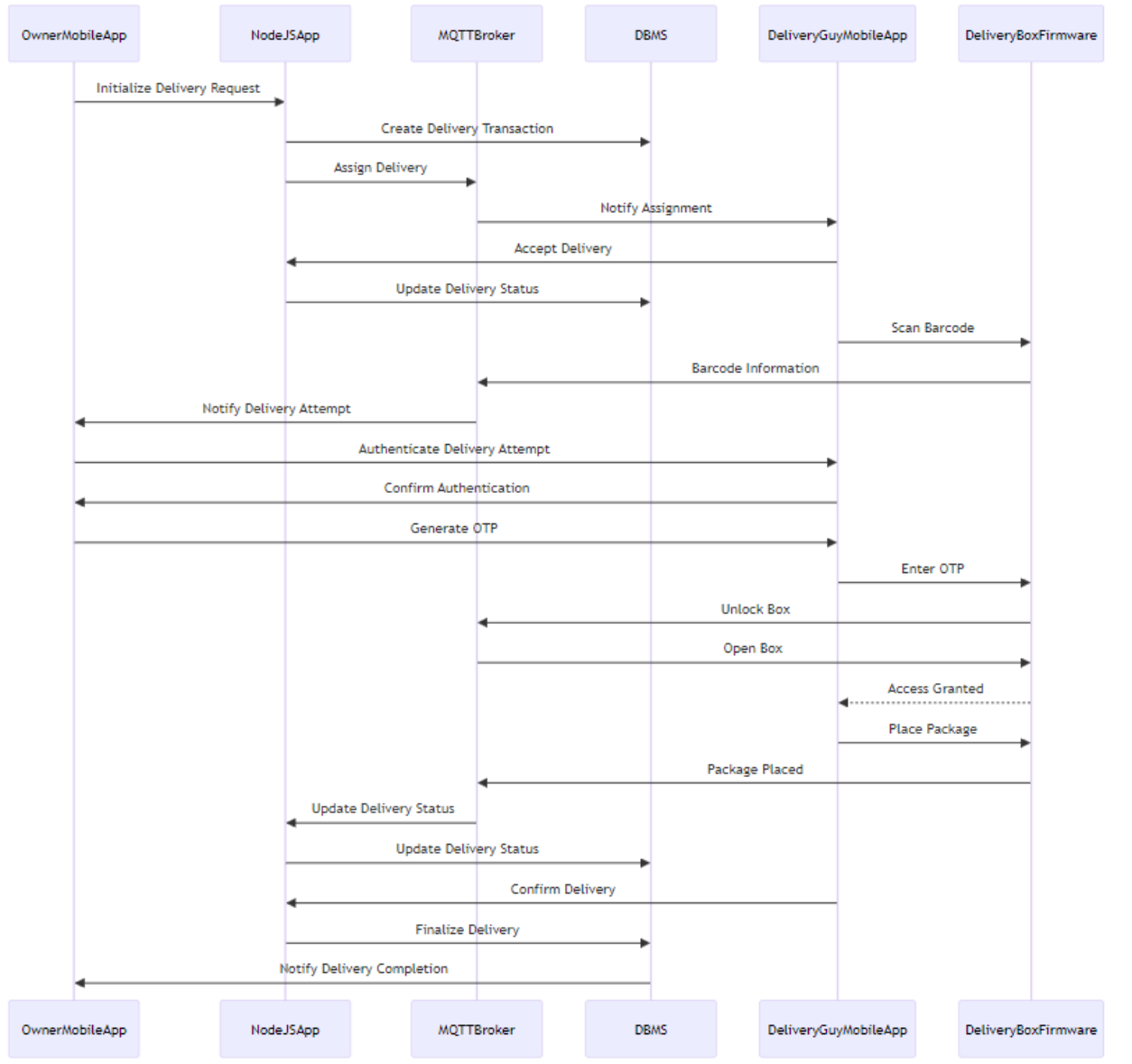
1. Initialization:
    - The Node.js app initializes and sets up the HTTP server, connecting to the MQTT broker, and establishing a connection to the PostgreSQL database.
    - MQTT clients for delivery boxes, owner mobile apps, and delivery guy mobile apps connect to the MQTT broker.
2. Box Initialization:
    - When a delivery box is powered on, it establishes a connection to the MQTT broker and subscribes to relevant topics for receiving commands and updates.
    - The box sends a message to the MQTT broker to indicate its availability and readiness to accept deliveries.
3. Delivery Request:
    - An owner places a delivery request using the mobile app by providing package details and recipient information.
    - The mobile app sends a request to the Node.js app via HTTP, triggering the creation of a delivery transaction in the database.
4. Delivery Assignment:
    - The Node.js app receives the delivery request, validates the request, and assigns the delivery to an available delivery guy.
    - It updates the status of the delivery transaction in the database and notifies the assigned delivery guy via MQTT.
5. Delivery Notification:
    - The delivery guy receives a notification on their mobile app about the assigned delivery, including package details and recipient information.

- They acknowledge the notification, indicating their acceptance of the delivery assignment.

6. Delivery Process:
    - The delivery guy proceeds to the designated location with the package, scanning the barcode on the delivery box upon arrival.
    - The box firmware sends a message to the MQTT broker with the scanned barcode information.
7. Owner Verification:
    - The owner receives a notification on their mobile app about the delivery attempt, including details of the delivery guy and package.
    - They authenticate the delivery by verifying the delivery guy's identity and confirming the delivery attempt on their mobile app.
8. Package Access:
    - Upon authentication, the owner generates an OTP (One-Time Password) and shares it with the delivery guy.
    - The delivery guy enters the OTP into the delivery box interface, triggering the box to unlock and grant access to the package compartment.
9. Package Delivery:
    - The delivery guy places the package inside the box compartment, closes the box securely, and confirms the delivery on their mobile app.
    - The box firmware sends a message to the MQTT broker indicating the successful delivery, and the delivery guy updates the delivery status in the database via the Node.js app.
10. Delivery Completion:
    - The owner receives a final notification on their mobile app confirming the successful delivery of the package.
    - The delivery transaction is marked as completed in the database, and relevant records are updated accordingly.

This sequence illustrates the seamless flow of interactions among MQTT, DBMS, Node.js app, owner mobile app, and delivery guy mobile app, ensuring efficient package delivery management and tracking.

This following diagram illustrates the flow of interactions between the Owner Mobile App, Node.js App, MQTT Broker, Delivery Guy Mobile App, Delivery Box Firmware, and DBMS during the initialization phase and the delivery process.


OwnerMobileApp
NodeJSApp
MQTTBroker
DBMS
DeliveryGuyMobileApp
DeliveryBoxFirmware
Initialize Delivery Request
Create Delivery Transaction
Assign Delivery
Notify Assignment
Accept Delivery
Update Delivery Status
Scan Barcode
Barcode Information
Notify Delivery Attempt
Authenticate Delivery Attempt
Confirm Authentication
Generate OTP
Enter OTP
Unlock Box
Open Box
Access Granted
Place Package
Package Placed
Update Delivery Status
Update Delivery Status
Confirm Delivery
Finalize Delivery
Notify Delivery Completion
OwnerMobileApp
NodeJSApp
MQTTBroker
DBMS
DeliveryGuyMobileApp
DeliveryBoxFirmware

AHLN Project Software

1. **Initiate Delivery Request**:
    - The process starts when a delivery request is initiated, either by a customer placing an order or by a vendor scheduling a delivery.
2. **Assign Delivery ID**:
    - A unique Delivery ID is assigned to the delivery request, which will be used to track the delivery process.
3. **Prepare Package**:
    - The package is prepared, including packaging the items securely and generating any necessary labels or barcodes.
4. **Schedule Pickup**:
    - The pickup of the package from the vendor's location is scheduled. This involves coordinating with the vendor to ensure the package is ready for pickup.
5. **Pickup Package**:
    - The delivery personnel pick up the package from the vendor's location, verifying the contents and ensuring proper documentation.
6. **Transport Package**:
    - The package is transported to the destination address, either directly or through intermediate distribution centers.
7. **Arrive for Delivery**:
    - The delivery personnel arrive at the destination address to deliver the package.
8. **Attempt Delivery**:
    - The delivery personnel attempt to deliver the package to the customer.
9. **Customer Not Available**:
    - If the customer is not available to receive the package, the delivery personnel contact the customer using the provided contact information.
10. **Authentication Process**:
    - The customer provides an OTP to the delivery personnel to authenticate the delivery.
11. **OTP Verification**:
    - The delivery personnel enter the OTP received from the customer into the delivery box's interface.
12. **Unlock Delivery Box**:
    - Upon successful verification of the OTP, the delivery box is unlocked, allowing the delivery personnel to place the package inside.
13. **Place Package**:
    - The delivery personnel place the package inside the delivery box.
14. **Close Delivery Box**:
    - After placing the package inside, the delivery box is closed and locked securely.
15. **Update Delivery Status**:
    - The delivery status is updated in the system to reflect the current state of the delivery (e.g., "Delivered").

16. **Finalize Delivery**:
    - Once the delivery process is completed, the delivery is marked as finalized in the system.

This process flow incorporates the steps involved in authenticating the delivery using an OTP before placing the package inside the delivery box. It ensures that the delivery is secure and authorized even when the customer is not available to receive the package in person.

To handle communication between the delivery box firmware and the customer app via MQTT, you can implement MQTT topics and messages for various operations such as package delivery notifications, authentication requests, and package status updates. Below is an outline of how you can structure the MQTT communication:

1. **Package Delivery Notification**:
    - When the delivery guy arrives at the customer's home and attempts to deliver the package, the delivery box firmware publishes a message to a specific MQTT topic indicating the delivery attempt.
    - Example MQTT Topic: **`delivery/notifications`**
    - Example Message: **`{"delivery_id": 123, "status": "delivery_attempted"}`**
2. **Authentication Request**:
    - If the customer is not at home to receive the package, the delivery guy initiates an authentication request by scanning the barcode on the delivery box.
    - The delivery box firmware publishes an authentication request message to a specific MQTT topic.
    - Example MQTT Topic: **`delivery/authentication`**
    - Example Message: **`{"delivery_id": 123, "barcode": "ABC123"}`**
3. **Customer Response**:
    - Upon receiving the authentication request, the customer app receives a notification and prompts the customer to send an authentication response (OTP) to the delivery guy.
    - The customer app subscribes to a specific MQTT topic for authentication responses.
    - Example MQTT Topic: **`delivery/authentication/response`**
    - Example Message: **`{"delivery_id": 123, "otp": "123456"}`**
4. **Package Status Update**:
    - Once the delivery guy receives the OTP from the customer, he enters it into the delivery box.
    - The delivery box firmware validates the OTP and updates the package status.
    - The delivery box firmware publishes a package status update message to a specific MQTT topic.
    - Example MQTT Topic: **`delivery/status`**
    - Example Message: **`{"delivery_id": 123, "status": "delivered"}`**

By using MQTT for communication between the delivery box firmware and the customer app, you can establish a reliable and real-time messaging system for package delivery notifications, authentication requests, and package status updates. Ensure that both the delivery box firmware and the customer app are configured to publish and subscribe to the appropriate MQTT topics to facilitate seamless communication.

The DBMS acting as middleware between the delivery box firmware and the customer app, we can enhance the procedures to handle the communication flow more efficiently. Here are some suggestions for improving the stored procedures:

1. **Package Delivery Notification**:
    - When the delivery box firmware detects a delivery attempt, it can call a stored procedure in the DBMS to record the delivery attempt and notify the customer app.
    - The stored procedure can update the delivery status in the database and publish a message to an MQTT topic to notify the customer app.
2. **Authentication Request**:
    - When the delivery box firmware sends an authentication request, it can invoke a stored procedure in the DBMS to record the request and notify the customer app.
    - The stored procedure can generate a unique authentication token (OTP) and associate it with the delivery in the database. It can then publish a message to an MQTT topic to request the customer's response.
3. **Customer Response**:
    - When the customer app sends an authentication response (OTP), it can trigger a stored procedure in the DBMS to validate the OTP and update the delivery status accordingly.
    - The stored procedure can verify the OTP against the one stored in the database for the corresponding delivery. If the OTP is valid, it can update the delivery status and notify the delivery box firmware to proceed with package delivery.
4. **Package Status Update**:
    - After successfully authenticating the customer's response, the delivery box firmware can call a stored procedure in the DBMS to update the package status and complete the delivery process.
    - The stored procedure can update the delivery status in the database and publish a message to an MQTT topic to notify the customer app of the delivery completion.

By integrating the stored procedures with the delivery box firmware and the customer app, the DBMS can act as a central middleware component to manage the communication flow between the two endpoints. This approach ensures data consistency, reliability, and real-time updates throughout the package delivery process.

STORED PROCEDURES

Insert Customer Procedure:

- This procedure inserts a new customer into the **customer** table.

```
CREATE OR REPLACE PROCEDURE insert_customer(
   p_name VARCHAR(255),
   p_address VARCHAR(255),
   p_gr VARCHAR(255),
   p_box_id INT,
   p_datetime_purchased_box2 TIMESTAMP,
   p_box_id2 INT,
   p_datetime_purchased_box2 TIMESTAMP,
   p_phone_no VARCHAR(20),
   p_alt_phone_no VARCHAR(20),
   p_payment_method VARCHAR(50)
)
AS $$
BEGIN
   INSERT INTO customer (Name, Address, GR, BoxID, DateTimePurchasedBox2, BoxID2,
DateTimePurchasedBox2, PhoneNo, AltPhoneNo, PamentMethod)
   VALUES (p_name, p_address, p_gr, p_box_id, p_datetime_purchased_box2, p_box_id2,
p_datetime_purchased_box2, p_phone_no, p_alt_phone_no, p_payment_method);
END;
$$ LANGUAGE plpgsql;
```

2. **Insert Box Procedure**:
    - This procedure inserts a new box into the **box** table.

```
CREATE OR REPLACE PROCEDURE insert_box(
   p_ser_no VARCHAR(50),
   p_n_compartments INT,
   p_compartment1_filled BOOLEAN,
   p_compartment2_filled BOOLEAN,
   p_compartment3_filled BOOLEAN,
   p_video_id INT
)
AS $$
BEGIN
```

```
   INSERT INTO box (SerNo, N_Compartmrnts, Compartment1_Filled, Compartment2_Filled,
Compartment3_Filled, VideoID)
   VALUES (p_ser_no, p_n_compartments, p_compartment1_filled, p_compartment2_filled,
p_compartment3_filled, p_video_id);
END;
$$ LANGUAGE plpgsql;
```

Insert Vendor Procedure:

- This procedure inserts a new vendor into the **vendor** table.

```
CREATE OR REPLACE PROCEDURE insert_vendor(
   p_name VARCHAR(255),
   p_address VARCHAR(255),
   p_url VARCHAR(255),
   p_api_ref VARCHAR(255),
   p_phone_no VARCHAR(20)
)
AS $$
BEGIN
   INSERT INTO vendor (name, Address, URL, API_ref, PhoneNo)
   VALUES (p_name, p_address, p_url, p_api_ref, p_phone_no);
END;
$$ LANGUAGE plpgsql;
```

Insert Delivery Procedure:

- This procedure inserts a new delivery into the **delivery** table.

```
CREATE OR REPLACE PROCEDURE insert_delivery(
   p_date_time TIMESTAMP,
   p_bar_code VARCHAR(50),
   p_qr_code VARCHAR(50),
   p_from_id INT,
   p_to_customer_id INT,
   p_delivered BOOLEAN
)
AS $$
BEGIN
   INSERT INTO delivery (DateTime, BarCode, QR_Code, From_ID, to_CustomerID, Delivered)
   VALUES (p_date_time, p_bar_code, p_qr_code, p_from_id, p_to_customer_id, p_delivered);
END;
$$ LANGUAGE plpgsql;
```

Insert Operation Procedure:

- This procedure inserts a new operation into the **operations** table.

```
CREATE OR REPLACE PROCEDURE insert_operation(
   p_delivery_id INT,
   p_state_method VARCHAR(50)
)
AS $$
BEGIN
   INSERT INTO operations (DeliveryID, State_method)
   VALUES (p_delivery_id, p_state_method);
END;
$$ LANGUAGE plpgsql;
```

Here's a Python code snippet that demonstrates how to handle MQTT send/receive using the Paho MQTT client library and call the stored procedures you provided:

```
import paho.mqtt.client as mqtt
import psycopg2
from datetime import datetime

# MQTT configurations
MQTT_BROKER = "mqtt.example.com"
MQTT_PORT = 1883
MQTT_TOPIC = "delivery"

# PostgreSQL configurations
DB_HOST = "localhost"
DB_NAME = "your_database"
DB_USER = "your_username"
DB_PASSWORD = "your_password"

# Connect to PostgreSQL database
conn = psycopg2.connect(host=DB_HOST, database=DB_NAME, user=DB_USER,
password=DB_PASSWORD)
cur = conn.cursor()

# MQTT message callback
def on_message(client, userdata, msg):
   payload = msg.payload.decode("utf-8")
   print(f"Received message: {payload}")

   # Parse payload and call appropriate stored procedure
   if payload == "new_delivery":
      # Call stored procedure to insert delivery
      insert_delivery(datetime.now(), "123456", "789012", 1, 2, False)
   elif payload == "update_delivery_status":
      # Call stored procedure to update delivery status
      update_delivery_status(1, "in_transit")
```

```
# Connect to MQTT broker
client = mqtt.Client()
client.connect(MQTT_BROKER, MQTT_PORT)

# Subscribe to MQTT topic
client.subscribe(MQTT_TOPIC)
client.on_message = on_message

# Callback functions for stored procedures
def insert_delivery(date_time, bar_code, qr_code, from_id, to_customer_id, delivered):
    try:
        cur.callproc("insert_delivery", (date_time, bar_code, qr_code, from_id, to_customer_id, delivered))
        conn.commit()
        print("Delivery inserted successfully")
    except psycopg2.Error as e:
        print(f"Error inserting delivery: {e}")

def update_delivery_status(delivery_id, state_method):
    try:
        cur.callproc("insert_operation", (delivery_id, state_method))
        conn.commit()
        print("Delivery status updated successfully")
    except psycopg2.Error as e:
        print(f"Error updating delivery status: {e}")

# Start MQTT client loop
client.loop_forever()
```

In this code:

- We establish connections to both the MQTT broker and the PostgreSQL database.
- The **on_message** function serves as the callback for incoming MQTT messages. Depending on the payload received, it calls the appropriate stored procedure.
- Stored procedure functions (**insert_delivery** and **update_delivery_status**) are defined to execute the respective stored procedures in the database.
- When a message is received on the MQTT topic, the **on_message** function is invoked to handle it.

Make sure to replace the placeholders with your actual MQTT broker and database configurations. Also, ensure that you have the Paho MQTT client library and psycopg2 installed (**pip install paho-mqtt psycopg2**).

Node.js code

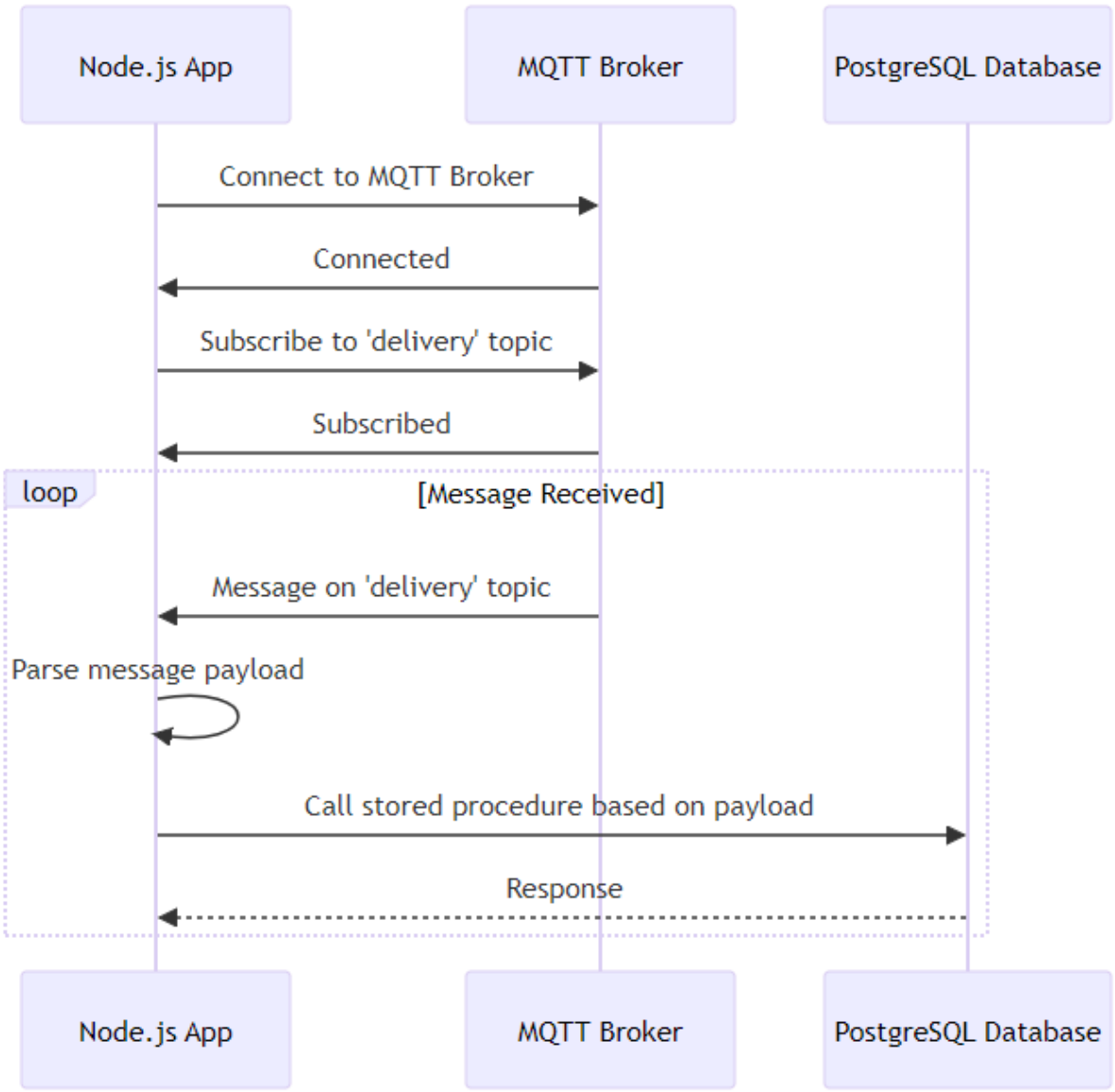
```
const mqtt = require('mqtt');
const { Pool } = require('pg');

// MQTT configurations
const MQTT_BROKER = 'mqtt://mqtt.example.com';
const MQTT_TOPIC = 'delivery';

// PostgreSQL configurations
const pool = new Pool({
  user: 'your_username',
  host: 'localhost',
  database: 'your_database',
  password: 'your_password',
  port: 5432,
});

// Connect to MQTT broker
const client = mqtt.connect(MQTT_BROKER);

// Subscribe to MQTT topic
client.on('connect', () => {
  console.log('Connected to MQTT broker');
  client.subscribe(MQTT_TOPIC);
});

// MQTT message handler
client.on('message', async (topic, message) => {
  if (topic === MQTT_TOPIC) {
    const payload = message.toString();
    console.log(`Received message: ${payload}`);

    // Parse payload and call appropriate stored procedure
    if (payload === 'new_delivery') {
      try {
        await insertDelivery(new Date(), '123456', '789012', 1, 2, false);
        console.log('Delivery inserted successfully');
      } catch (error) {
        console.error('Error inserting delivery:', error);
      }
    } else if (payload === 'update_delivery_status') {
      try {
        await updateDeliveryStatus(1, 'in_transit');
        console.log('Delivery status updated successfully');
      } catch (error) {
```

```
      console.error('Error updating delivery status:', error);
    }
   }
  }
});

// Function to insert delivery using stored procedure
async function insertDelivery(date_time, bar_code, qr_code, from_id, to_customer_id, delivered) {
  const client = await pool.connect();
  try {
    await client.query('BEGIN');
    await client.query('CALL insert_delivery($1, $2, $3, $4, $5, $6)', [date_time, bar_code, qr_code,
from_id, to_customer_id, delivered]);
    await client.query('COMMIT');
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

// Function to update delivery status using stored procedure
async function updateDeliveryStatus(delivery_id, state_method) {
  const client = await pool.connect();
  try {
    await client.query('BEGIN');
    await client.query('CALL update_delivery_status($1, $2)', [delivery_id, state_method]);
    await client.query('COMMIT');
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}
```

In this code:

- We establish connections to both the MQTT broker and the PostgreSQL database.
- Upon connecting to the MQTT broker, we subscribe to the specified topic (**delivery**).
- When a message is received on the MQTT topic, the message handler function is invoked to parse the payload and call the appropriate stored procedure in the database.
- Stored procedure functions (**insertDelivery** and **updateDeliveryStatus**) are defined to execute the respective stored procedures in the database.
- We use the **pg** library to interact with the PostgreSQL database.

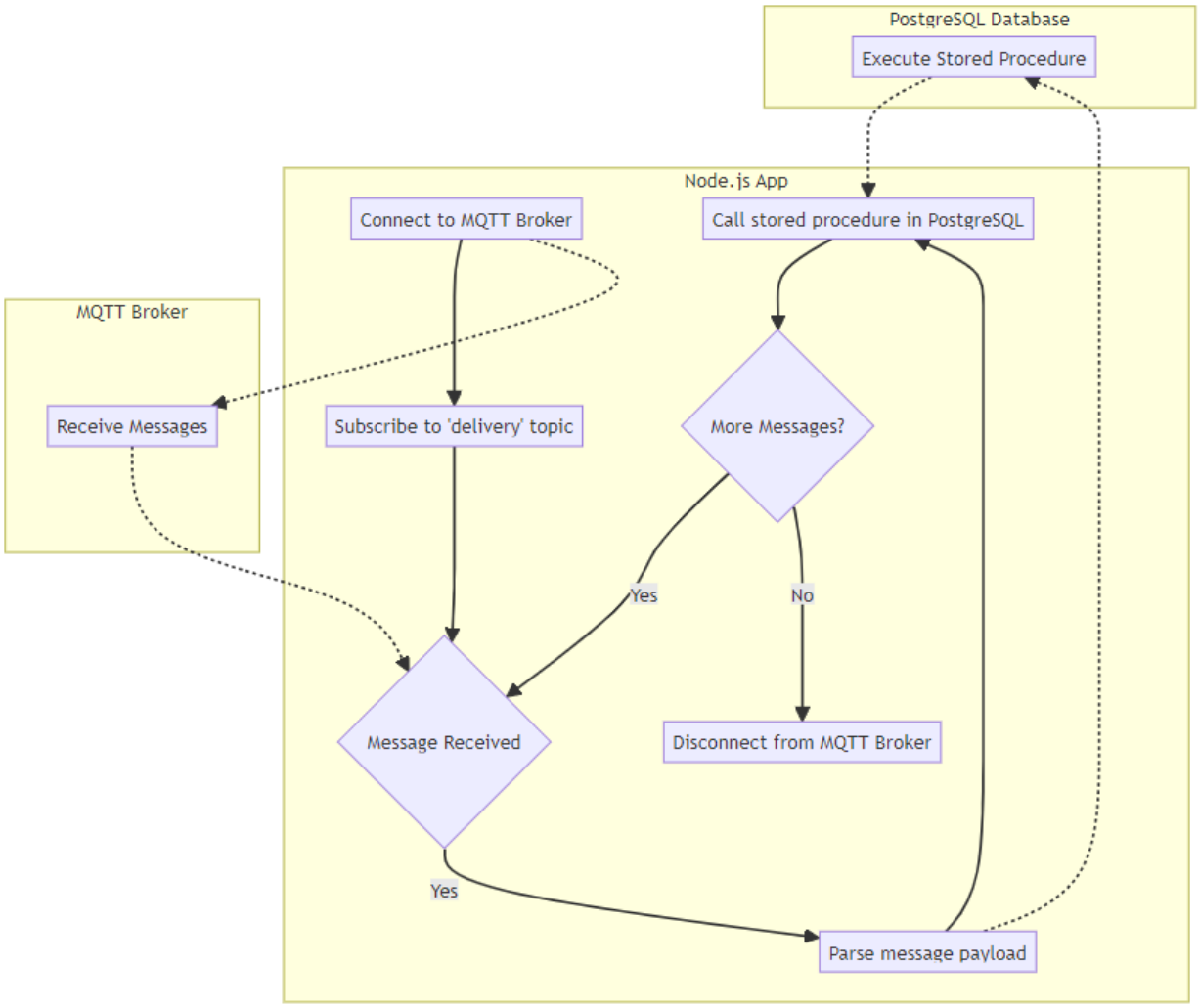
- Make sure to replace the placeholders with your actual MQTT broker and database configurations. Also, ensure that you have the required libraries installed (`npm install mqtt pg`).

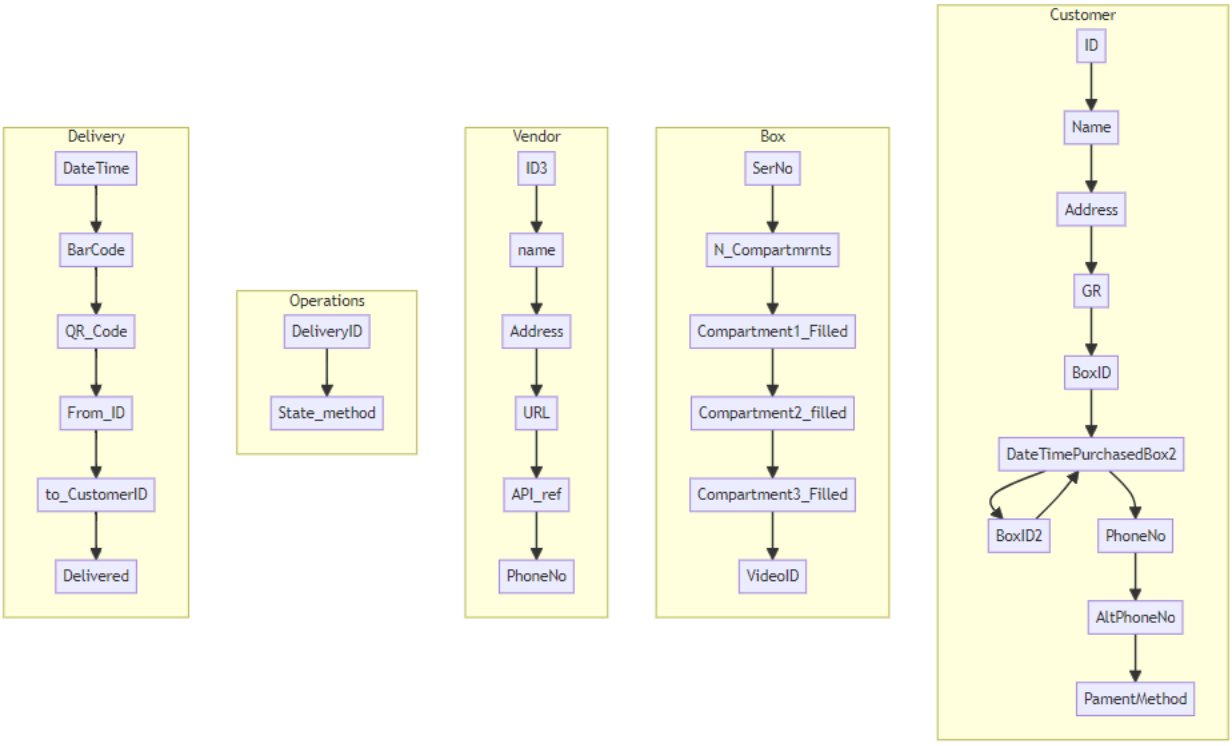


- The Node.js application (Client) connects to the MQTT broker (MQTTBroker).
- After connecting, it subscribes to the 'delivery' topic on the MQTT broker.
- When a message is received on the 'delivery' topic, the Node.js application parses the payload and calls the appropriate stored procedure in the PostgreSQL database (PostgreSQL).
- Finally, the PostgreSQL database responds to the Node.js application after executing the stored procedure.

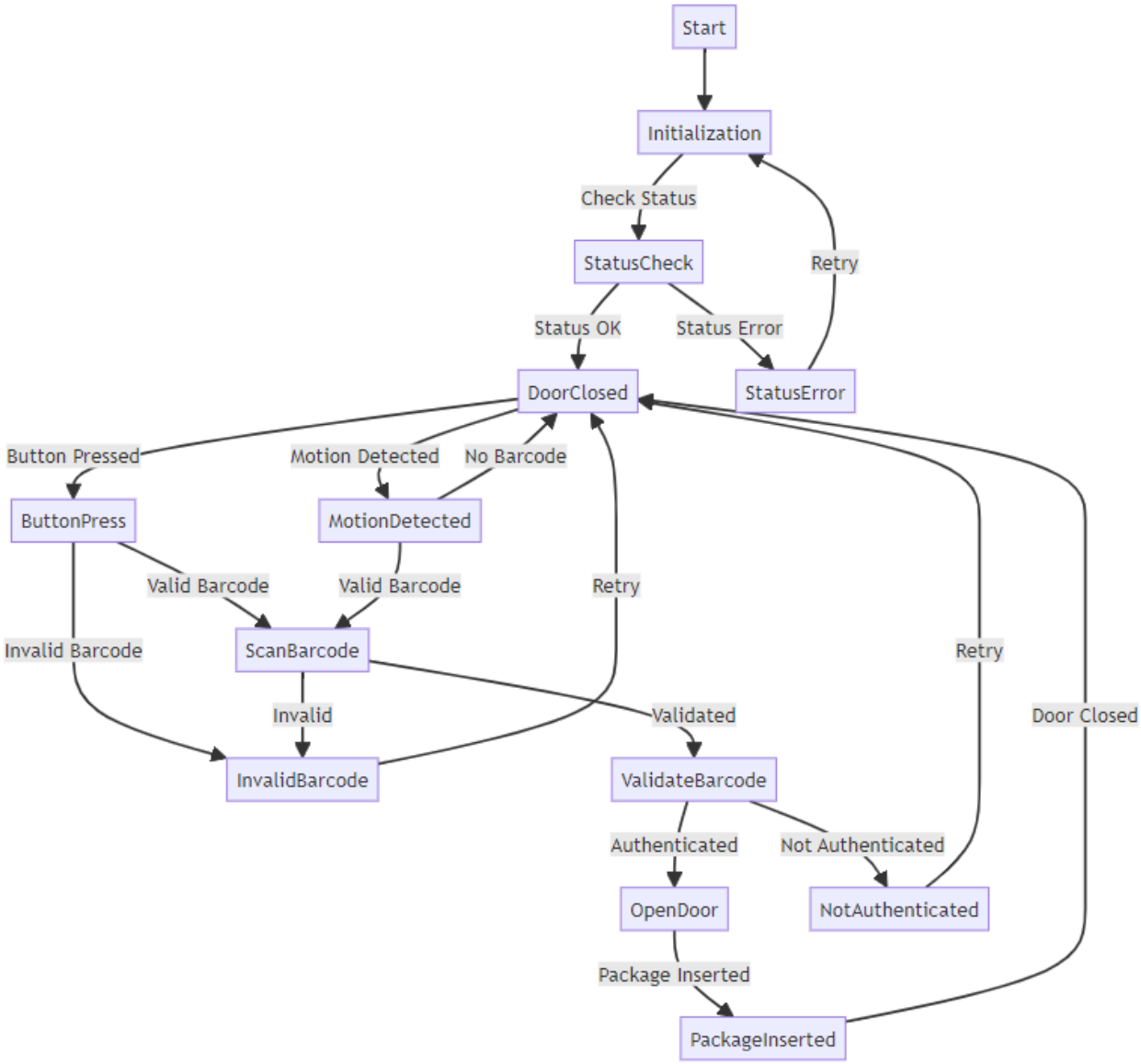
```
sequenceDiagram
    participant Client as Node.js App
    participant MQTTBroker as MQTT Broker
    participant PostgreSQL as PostgreSQL Database

    Client->>MQTTBroker: Connect to MQTT Broker
    MQTTBroker->>Client: Connected
    Client->>MQTTBroker: Subscribe to 'delivery' topic
    MQTTBroker->>Client: Subscribed

    loop Message Received
        MQTTBroker->>Client: Message on 'delivery' topic
        Client->>Client: Parse message payload
        Client->>PostgreSQL: Call stored procedure based on payload
        PostgreSQL-->>Client: Response
    end
```

Dataflow Diagram




Delivery
DateTime
BarCode
QR_Code
From_ID
to_CustomerID
Delivered
Operations
DeliveryID
State_method
Vendor
ID3
name
Address
URL
API_ref
PhoneNo
Box
SerNo
N_Compartmrnts
Compartment1_Filled
Compartment2_filled
Compartment3_Filled
VideoID
Customer
ID
Name
Address
GR
BoxID
DateTimePurchasedBox2
BoxID2
PhoneNo
AltPhoneNo
PamentMethod

Firm ware



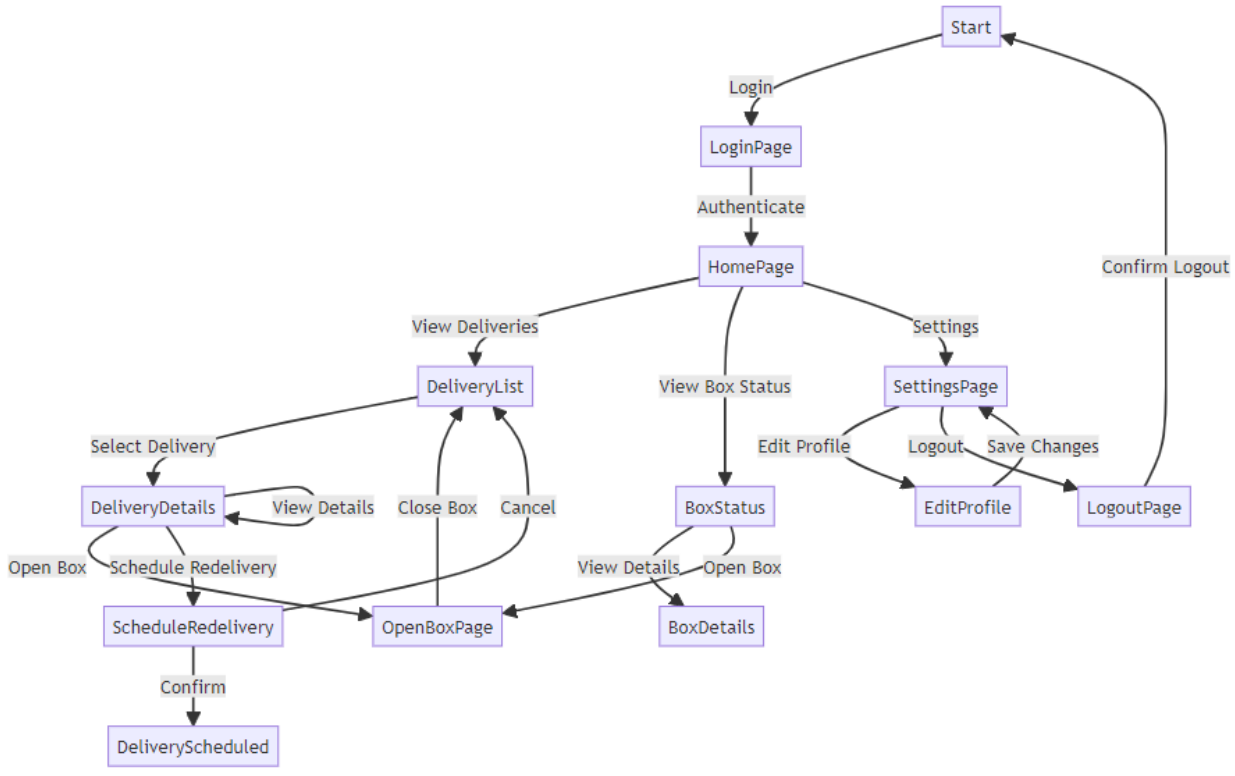
1. initialization of the firmware.
2. Checking the status of the box.
3. If the status is OK, the door remains closed.
4. If the button is pressed or motion is detected:
    - If a valid barcode is scanned, it is validated.
    - If the barcode is invalid, the process returns to the door closed state.
5. If the barcode is validated and authenticated, the door opens.
6. After the package is inserted, the door is closed again.
7. If there's an error in the status check, the firmware retries initialization.

graph TD;

```
Start --> Initialization;
Initialization -->|Check Status| StatusCheck;
StatusCheck -->|Status OK| DoorClosed;
StatusCheck -->|Status Error| StatusError;
DoorClosed -->|Button Pressed| ButtonPress;
DoorClosed -->|Motion Detected| MotionDetected;
ButtonPress -->|Valid Barcode| ScanBarcode;
ButtonPress -->|Invalid Barcode| InvalidBarcode;
ScanBarcode -->|Validated| ValidateBarcode;
ScanBarcode -->|Invalid| InvalidBarcode;
ValidateBarcode -->|Authenticated| OpenDoor;
ValidateBarcode -->|Not Authenticated| NotAuthenticated;
NotAuthenticated -->|Retry| DoorClosed;
OpenDoor -->|Package Inserted| PackageInserted;
PackageInserted -->|Door Closed| DoorClosed;
InvalidBarcode -->|Retry| DoorClosed;
MotionDetected -->|Valid Barcode| ScanBarcode;
MotionDetected -->|No Barcode| DoorClosed;
StatusError -->|Retry| Initialization;
```

Owner mobile app

Owner Mobile App Diagram

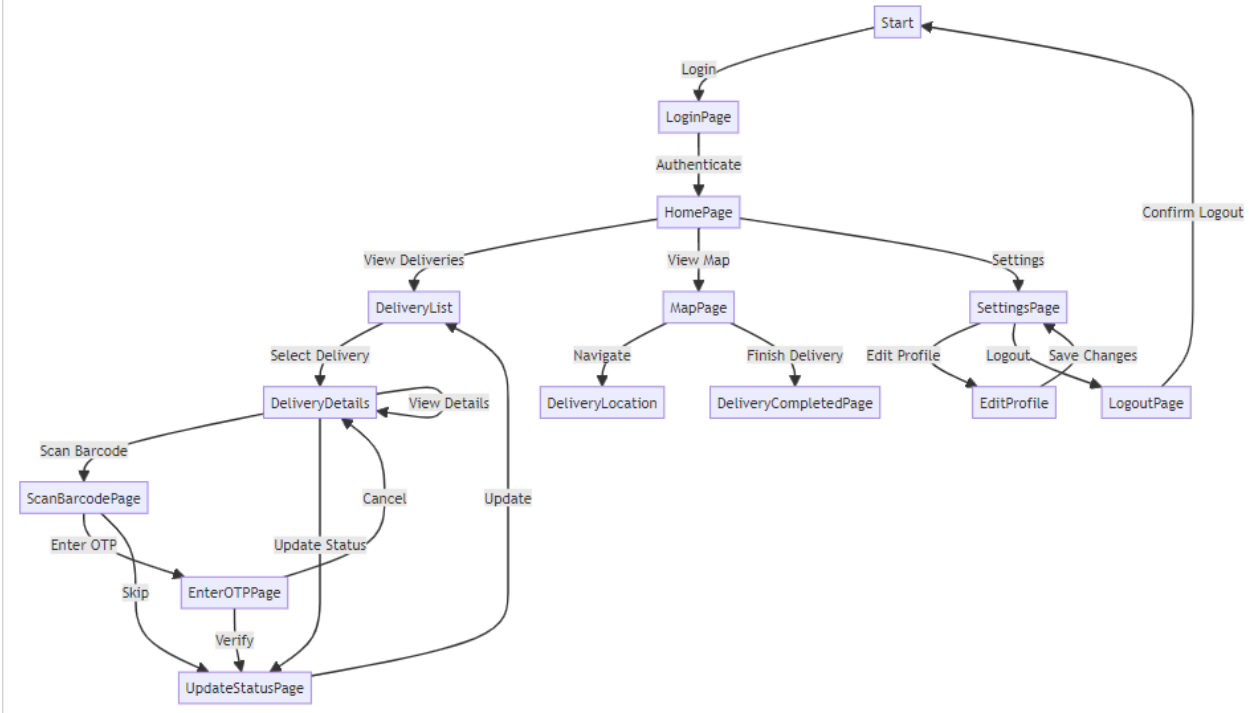


1. Start: Beginning of the app.
2. Login Page: User authentication.

3. Home Page: Main dashboard with options to view deliveries, box status, and settings.
4. Delivery List: List of deliveries for the user.
5. Delivery Details: Details of a selected delivery, with options to view details, open the box, or schedule a redelivery.
6. Box Status: Status of the delivery box, with options to view details or open the box.
7. Settings Page: Options to edit profile or logout.
8. Edit Profile: Page to edit user profile information.
9. Schedule Redelivery: Page to schedule a redelivery of a package.
10. Open Box Page: Page to open the delivery box.
11. Logout Page: Confirmation page for logout.
12. Flow loops back to Start after logout confirmation.

```
graph TD;
    Start[Start] -->|Login| LoginPage;
    LoginPage -->|Authenticate| HomePage;
    HomePage -->|View Deliveries| DeliveryList;
    HomePage -->|View Box Status| BoxStatus;
    HomePage -->|Settings| SettingsPage;
    DeliveryList -->|Select Delivery| DeliveryDetails;
    DeliveryDetails -->|View Details| DeliveryDetails;
    DeliveryDetails -->|Open Box| OpenBoxPage;
    DeliveryDetails -->|Schedule Redelivery| ScheduleRedelivery;
    BoxStatus -->|View Details| BoxDetails;
    BoxStatus -->|Open Box| OpenBoxPage;
    SettingsPage -->|Edit Profile| EditProfile;
    SettingsPage -->|Logout| LogoutPage;
    EditProfile -->|Save Changes| SettingsPage;
    ScheduleRedelivery -->|Confirm| DeliveryScheduled;
    ScheduleRedelivery -->|Cancel| DeliveryList;
    OpenBoxPage -->|Close Box| DeliveryList;
    LogoutPage -->|Confirm Logout| Start;
```

Delivery guy Mobile App Diagram



1. Start: Beginning of the app.
2. Login Page: User authentication.
3. Home Page: Main dashboard with options to view deliveries, map, and settings.
4. Delivery List: List of deliveries assigned to the delivery guy.
5. Delivery Details: Details of a selected delivery, with options to view details, scan barcode, or update status.
6. Map Page: Map showing the delivery location and navigation options.
7. Settings Page: Options to edit profile or logout.
8. Edit Profile: Page to edit user profile information.
9. Scan Barcode Page: Page to scan the barcode on the delivery box.
10. Enter OTP Page: Page to enter the OTP sent by the customer.
11. Update Status Page: Page to update the status of the delivery.
12. Logout Page: Confirmation page for logout.
13. Flow loops back to Start after logout confirmation.

```
graph TD;

    Start[Start] -->|Login| LoginPage;
    LoginPage -->|Authenticate| HomePage;
    HomePage -->|View Deliveries| DeliveryList;
    HomePage -->|View Map| MapPage;
    HomePage -->|Settings| SettingsPage;
    DeliveryList -->|Select Delivery| DeliveryDetails;
```

```
DeliveryDetails -->|View Details| DeliveryDetails;
DeliveryDetails -->|Scan Barcode| ScanBarcodePage;
DeliveryDetails -->|Update Status| UpdateStatusPage;
MapPage -->|Navigate| DeliveryLocation;
MapPage -->|Finish Delivery| DeliveryCompletedPage;
SettingsPage -->|Edit Profile| EditProfile;
SettingsPage -->|Logout| LogoutPage;
EditProfile -->|Save Changes| SettingsPage;
ScanBarcodePage -->|Enter OTP| EnterOTPPage;
ScanBarcodePage -->|Skip| UpdateStatusPage;
EnterOTPPage -->|Verify| UpdateStatusPage;
EnterOTPPage -->|Cancel| DeliveryDetails;
UpdateStatusPage -->|Update| DeliveryList;
LogoutPage -->|Confirm Logout| Start;
```