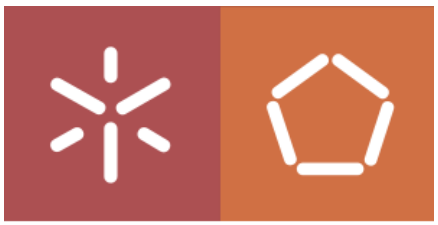

Universidade do Minho
School of Engineering

Master's degree in Industrial Electronic Engineering and Computers

Embedded Systems

NexiPass - Smart solution for real-time access and consumption management

Final Project

Bruno Jácome | PG60194
Pedro Rodrigues | PG56205

Project supervised by:
Professor Doutor Adriano Tavares

Universidade do Minho
2025/2026

Index

illustration index

Tables Index

1. Motivations

The development of this system is motivated by the need for various establishments to ensure better control over access, consumption management, and product feedback.

A common problem for services such as bars, nightclubs, hotels, municipal swimming pools, among others, is the management of admissions, consumption, and payments, which is carried out using manual or minimally automated methods.

These processes generate problems such as long queues, slower service, errors in consumption records, and stock losses. On the other hand, for managers, the absence of an integrated system makes it difficult to control the business, compromising the traceability of transactions, employee performance analysis, identification of monetary losses, and feedback on product consumption. For customers, the experience becomes slower, with longer waiting times and cash handling.

2. Project Statement

This project proposes the development of an integrated system based on NFC cards to securely, efficiently and automatically manage user entry, service use and exit in various establishments.

The system allows cards to be activated at the entrance, accurately records consumption or additional services, associates each transaction with the user and the person responsible for customer service, and controls payments automatically. Through immediate physical feedback such as lights and sound, the system clearly communicates the status of the card, as well as providing feedback on the products themselves (preferences, stock, etc.) and the total bill.

The main objective is to optimize operational processes, reduce manual errors, improve the user experience and optimize the management and analysis of the establishment.

Figure 1 - Illustrative representation of the NexiPass model to be implemented

2.1. Requirements

In this chapter, we will analyse how the project will be implemented in terms of requirements (functional or non-functional) and constraints (technical or non-technical).

This analysis is important because it dictates how the project will develop, as well as how the subsequent analysis and implementation phases will occur, using these requirements and constraints as guidelines for the final product.

2.1.1. Functional Requirements

Functional requirements are the specific behaviours and functions that the system must perform, i.e., they describe what the system does to fulfil the project's objective.

The functional requirements of this project are:

- Activate NFC cards at the entrance
- Record entries, consumption and additional services
- Associate each transaction with the user and the employee responsible.
- Calculate the total consumption associated with the card.
- Save card data in a central database for later analysis (stocks, best-selling products, etc.).
- Display the final amount to be paid at the exit and deactivate the card after payment is complete.

2.1.2. Non-Functional Requirements

Non-functional requirements do not concern what the system does, but rather how the system should be in terms of quality, performance, security, usability, among others. They are:

- Preserve data privacy
- Ensure a simple and intuitive interface for employees and managers
- Ensure that processes are relatively fast
- Support multiple users
- Allow software updates without losing existing data
- Ensure low consumption and low hardware costs

2.2. Constraints

When talking about project constraints, we refer to the limitations or mandatory conditions that affect the development of the system. Constraints do not indicate what the project does but define the limits within which the system must be developed, which may be technical or non-technical limits.

2.2.1. Technical Constraints

Technical restrictions are mandatory technologies, specific tools, defined hardware, and programming language to be used. This project has the following technical restrictions:

- Use the Raspberry Pi board;
- Use C++ object-oriented language;
- Generated a linux image in Buildroot;;
- Implement at least one device driver;
- Use PThreads for multithreading.
- Use UML base tools.

2.2.2. Non-Technical Constraints

As Non-technical constraints are organisational, human, and economic conditions, including:

- Groups of two people
- Limited budgets
- Deadline at the end of the semester

2.3. System Overviews

After defining the requirements and constraints, a simplified high-level view of the system was developed.

The system's central node is the Raspberry Pi 4 Model B, which interacts locally with the RFID/NFC module to activate, record, and validate card transactions. Immediate physical feedback is provided by an RGB LED and a buzzer. For remote management, the Raspberry Pi communicates wirelessly (Wi-Fi) with the Cloud, where data relating to automatic billing, consumption recording and transaction association are stored and processed. Users and management staff access this information through an application that also communicates via the Cloud, enabling data analysis, operational optimisation and payment control, as illustrated in Figure 2.

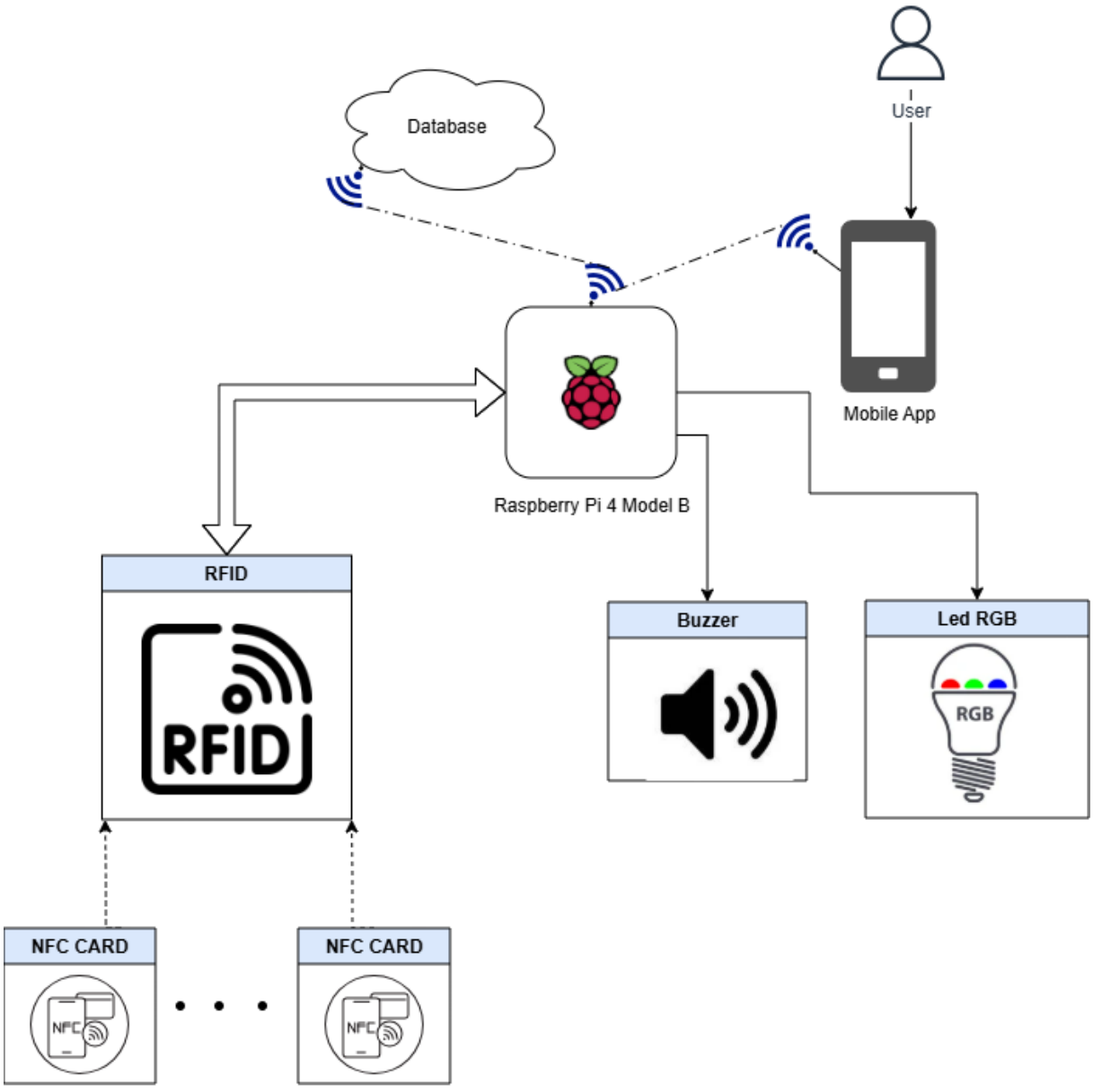

Figure 2 - System Overview

3. Analysis

During the Project Plan phase, a general, high-level introduction to the system to be developed was presented. Now, during the analysis phase, the goal is to provide more specific details on how the system works, to identify any problems and requirements that may not have been considered during the initial phase.

3.1. Market Study

Accompanied by technological growth and easy access to low-cost technologies, the market has sought to automate processes with the aim of developing solutions that require fewer human resources, are faster, and reduce the occurrence of errors. These technologies have also been used to collect relevant data, allowing market trends, productivity levels, and sales volumes to be monitored. Automation tools have been expanding over the last decade, and even greater exploitation is expected with the advancement of artificial intelligence technologies.

3.1.1. Similar products on the market

There are currently several projects on the market focused on NFC payments. However, these systems tend to focus exclusively on contactless payments, leaving aside complementary features such as inventory management, productivity control, and sales analysis, which ultimately depend on separate solutions.

MobiPag (Universidade do Minho):

Developed by the Algoritmi Center at the University of Minho, MobiPag is an integrated payment, ticketing, and coupon management system based on NFC technology, already implemented in real-world contexts. The system focuses mainly on payment and ticketing processes, but presents some challenges related to initial adoption and flexibility in the management of additional services, consumption, and integration with other systems, thus proving to be more oriented towards payment than complete operational management.

NOSIoT (NFC-based solution):

NOSIoT is a productivity and energy efficiency control solution that has already been implemented in real business contexts. It is a low-cost system that uses NFC tags in a “Tap & Go” model. It allows

employees to activate workstations only when they are at their workstations, thus avoiding unnecessary energy consumption. In addition to energy efficiency, the system also collects and stores productivity data in the cloud, recording which employee accessed the workstation and how long it remained in use.

3.1.2. Why does Nexipass stand out from these products?

After analyzing the market, we identified a significant gap in all-in-one systems for financial, stock, and productivity management.

In this regard, our product stands out as an integrated solution that not only allows you to manage payments but also provides different statistics that facilitate the monitoring of market trends and stock management. The goal is to address the limitations observed through features that allow:

- **Centralization of operational data:** automatic aggregation of all consumption records and movements by zone, eliminating the need for employees to operate multiple separate solutions.
- **Stock control and traceability of consumed items:** unlike conventional payment systems, our system automatically records and updates the number of products in stock.
- **Productivity management:** provides managers with employee performance indicators, such as speed, accuracy in consumption records, and ability to recommend new products.

This solution aims to resolve recurring problems, such as:

- **Human error:** manual recording is susceptible to errors, omissions, and duplications.
- **Complexity in consumption records:** operators need to switch between different interfaces (menu, card, POS) to close accounts, which increases operating time and the likelihood of error.
- **Lack of transparency for the customer:** the customer does not have real-time access to the number of consumptions made or the total amount to be debited.
- **Difficulty in obtaining customized reports:** managers need advanced reports (consumption by type, customer, or employee, analysis of peak hours and most used areas) without relying on data export between multiple systems and software.

3.2. System Architecture

Defining the system architecture is essential because it helps to define a structure, organization, and how all the blocks of a system interact with each other to ensure efficient and reliable operation. It serves as the master plan that guides the development and integration of hardware, software, and networks, ensuring compatibility, performance, and ease of maintenance.

3.2.1. Hardware Architecture

In the following image Figure 3, you can see the system's hardware architecture, which is reflected in all the system's physical components and how they interconnect with each other.

The architecture is based on and has as its main element Raspberry Pi, which is the main processing unit. It is responsible for processing data from the RFID readers and the user interface (APP), which will read the ID of the different NFC cards and the consumption recorded by employees on the interface. The Raspberry Pi is also responsible for acting on the outputs, namely the buzzer and the RGB LED.

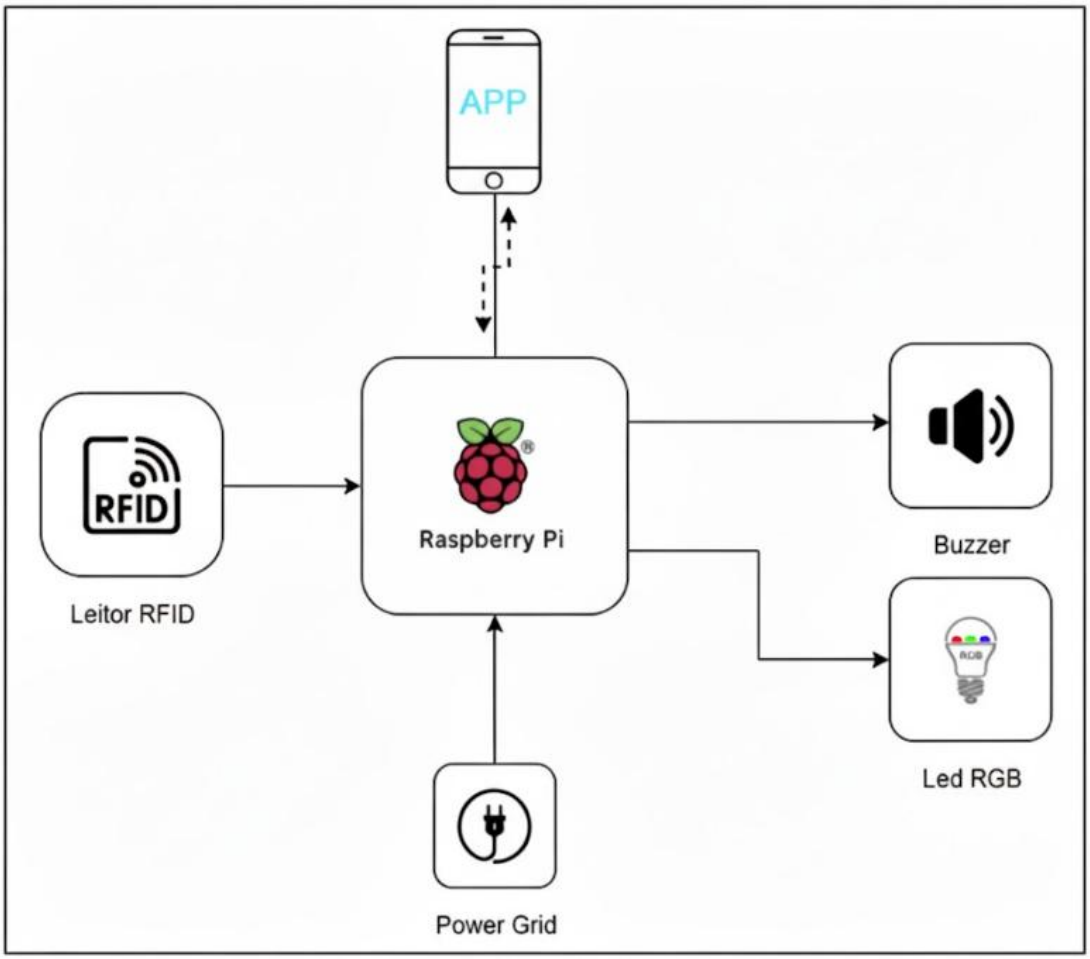

Figure 3 - Hardware Architecture

3.2.2. Software Architecture

Figure 4 illustrates the software architecture developed, organized into three main layers: Application, Middleware, and Linux.

This modular structure clearly defines the responsibilities of each component, from direct interaction with the hardware to data management and user interface.

The diagram also highlights the communication flow between the layers, ensuring a logical separation that facilitates system scalability, maintenance, and integration.

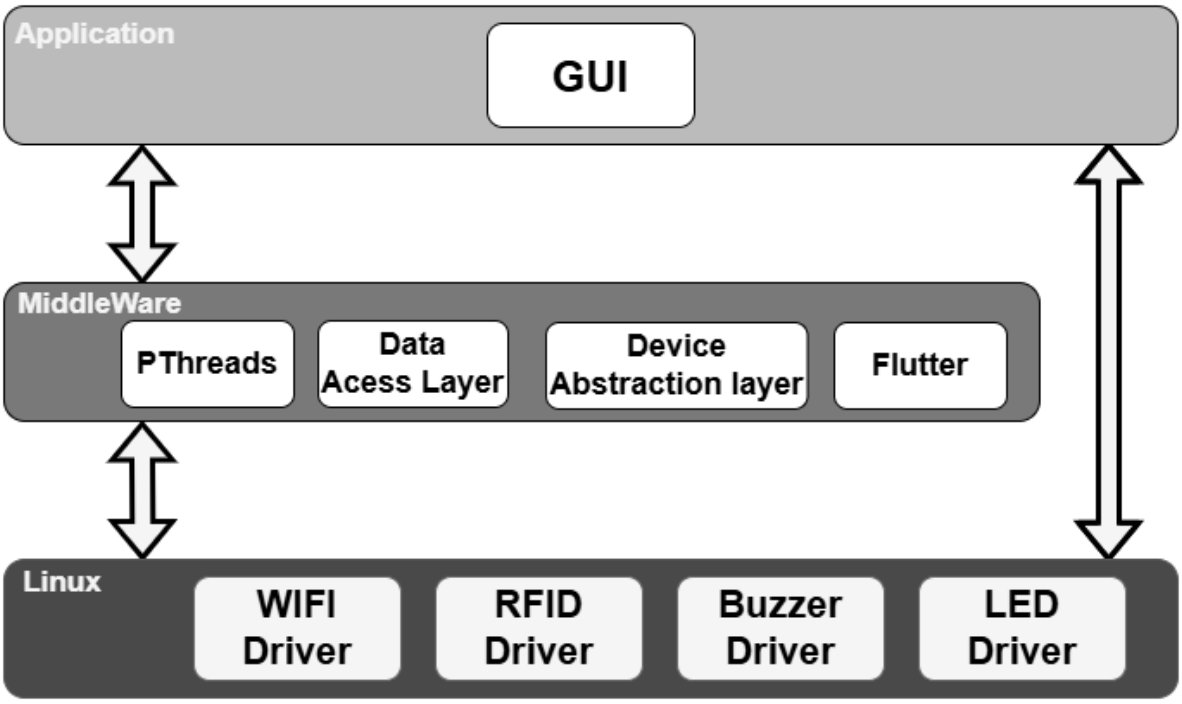

Figure 4 - Software Architecture

Software architecture is divided into three main layers, which are further divided into sublayers:

- **Custom Linux OS Layer:** This is the lowest layer, where the hardware drivers that communicate directly with the physical devices are located. It is the layer responsible for ensuring the interface between hardware and software.
    - **WIFI Driver**: manages wireless connectivity, necessary for wireless communication with the database.
    - **RFID Driver:** communicates with RFID card/tag readers.
    - **Buzzer Driver:** controls the buzzer sound for user feedback.
    - **LED Driver:** controls the LED for user feedback.

- **Middleware Layer:** Serves as a bridge between the application and the operating system (Linux), allowing the complexity of the drivers to be abstracted and facilitating application development.
    - **PThreads:** provides multithreading support.
    - **Data Access Layer:** layer that manages access to data for reading and writing to the database.
    - **Device Abstraction Layer:** isolates the application from specific device details.
    - **Flutter:** framework used to develop the interface and integrate the application
- **Application layer:** This is the layer visible to the user, where the main logic of the system runs.
    - **Graphical User Interface:** graphical interface that allows user interaction with the system.

3.3. System Analysis

Once the system architectures have been defined, it is essential to identify how the different components operate and interact with each other and how they engage with the outside world to fulfil the system's objective. At this stage, it is essential to evaluate the integration of the components to ensure that the system functions cohesively.

3.3.1. System Events

To accurately define the behaviour of a system, it is essential to have a detailed understanding of the events (Table 1) to which the system will be subject. In this sense, an inventory of these events was prepared to provide a rigorous and structured basis for the behavioural analysis of the system.

Event	Service Point	Source/Trigger	Type	System Response
NFC card activation	Entrance	Operator	Synchronous	Request phone number, register, and activate
Product/consumption registration	Point of sale	Operator	Synchronous	Validate card, add, register
Consumption/balance inquiry	Checkout/Point of Sale	Operator	Synchronous	Consult, calculate, and display
Card deactivation	cheackout	Operator	Synchronous	Deactivate and update
Exit authorization	Entrance	Operator	Synchronous	Check status, allow exit
Cloud synchronization	All points	System	Asynchronous	Send and update database

Table 1 - Event List

3.3.2. Use Cases Diagram

The use case diagram is an important tool in analysis, as it translates user and system needs into clear and visual functional requirements. Its simplicity and focus on the user perspective make it essential for aligning expectations and guiding the development of an effective and logically correct system.

In the diagram in Figure 5, the main actors are the Operator, the RFID sensor module and mobile application , and the external components (Buzzer, RGB LED, APP, and Database Engine).

The operator's first action is to select the service point (“Entrance,” “Point of sale,” or “Checkout”) on the application which will interact with the system. In any of the options, the next mandatory step is to read the NFC card (“read NFC card”) by the RFID sensor module, as highlighted in the diagram. After this reading, the operator has access only to the specific functionalities of the selected service point: card activation and exit authorization at the entrance; consumption registration at the point of sale; card deactivation and visualization of the total consumed at checkout.

The diagram uses the “include” and ‘extend’ relationships to show dependencies between use cases. For example, “activate card” is necessarily included in entry operations, while “add consumption” depends on the card being read beforehand.

It is also possible to see how the system interacts with the outside world and how these are activated according to the functionality performed. All relevant operations communicate with the Database Engine to record and obtain data from the DATABASE.

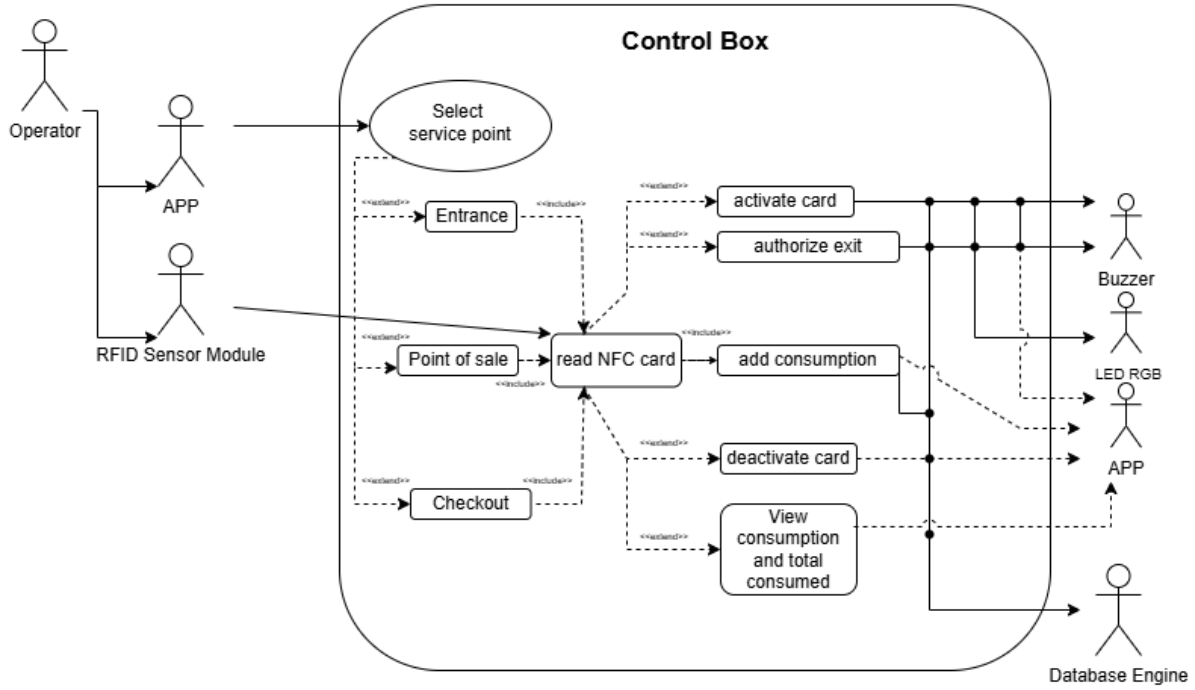

Figure 5 - Use cases Diagram

3.3.3. Sequence Diagrams

Sequence diagrams allow us to represent, in chronological order, the interaction between the different components of the NexiPass system during key operations.

These diagrams complement the conceptual model of the database, highlighting the flow of messages exchanged between the physical modules (Control Box, peripherals, database) and the interfaces used by the operator.

Three fundamental scenarios have been defined that correspond to the stages of system use:

1. Card Activation (Entrance)
2. Consumption Registration (Point of Sale)
3. Card Closure and Cleaning (Checkout)

Each diagram shows the role of the operator, the behaviour of the Control Box, and the interactions with the database and peripherals (application, RGB LED, and Buzzer), which provide real-time visual and audible feedback.

The diagram in Figure 6 represents the NFC card activation process in the NexiPass system.

When the operator approaches the card to the reader, the system reads the unique identifier (UID) and queries the database to verify the current status of the card.

If the card is deactivated (status=C), its status is updated to active (status=A), being temporarily associated with the user's mobile phone number and starting a new session of use.

At this point, the display shows the message “Card Activated,” the LED lights up green, and the buzzer emits a short sound, confirming successful activation.

If the card is already active, the system informs the operator with the message “Card already activated,” lighting up the blue LED as a visual indication of the status.

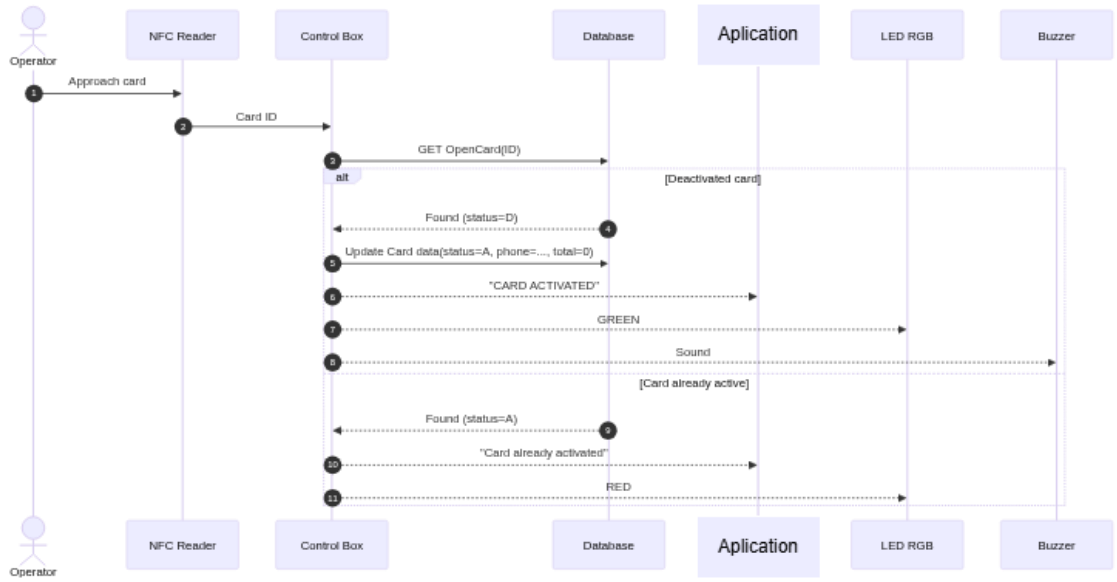

Figure 6 - Activation Sequence diagram

The diagram shown in Figure 7 illustrates the process of recording consumption during the use of an active card.

The operator selects the product and the respective quantity through the graphical user interface (GUI), which sends the data to the Control Box for validation of the card status.

If the card is active (status=A), a new record is created in the OpenConsumption table, and the accumulated total is updated in the OpenCard table.

The application shows the new total to be paid, the LED remains blue, and the buzzer emits a short sound, confirming the consumption record.

If the card is not active, the system prevents the record and notifies the operator with the message “Card not active,” accompanied by the red LED lighting up and a prolonged sound signalling the error.

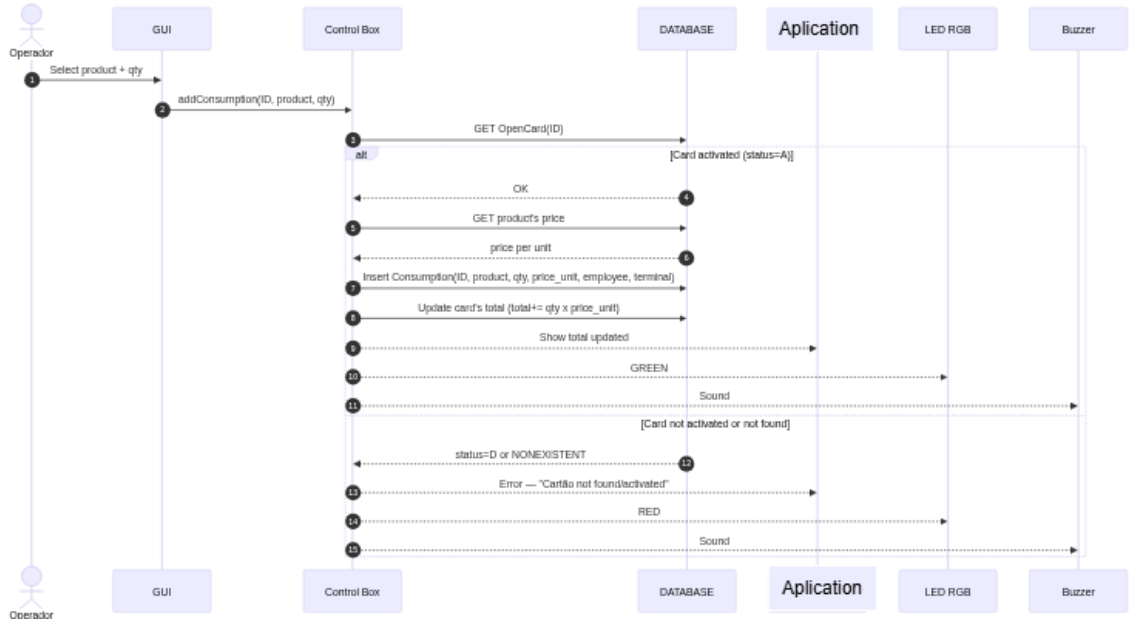

Figure 7 - Consumption Sequence diagram

Figure 8 shows the sequence diagram for closing the account and clearing the card.

When the operator confirms payment via the GUI, the Control Box requests the database for the consumption associated with the card in use.

If there is any recorded consumption, the system copies and aggregates this data into the ProductTotals analytical table, then deletes the corresponding records from the OpenConsumption table.

After this operation, the mobile phone number is removed and the card status is changed to status=D, indicating that it has been deactivated and is ready to be reused.

If there is no consumption, the system performs the cleaning process on the card, ensuring that it is reset to its initial state.

The display shows the message “Payment completed” (or “No consumption,” as applicable), the LED lights up green, and the buzzer emits a double or short sound, confirming the completion of the process.

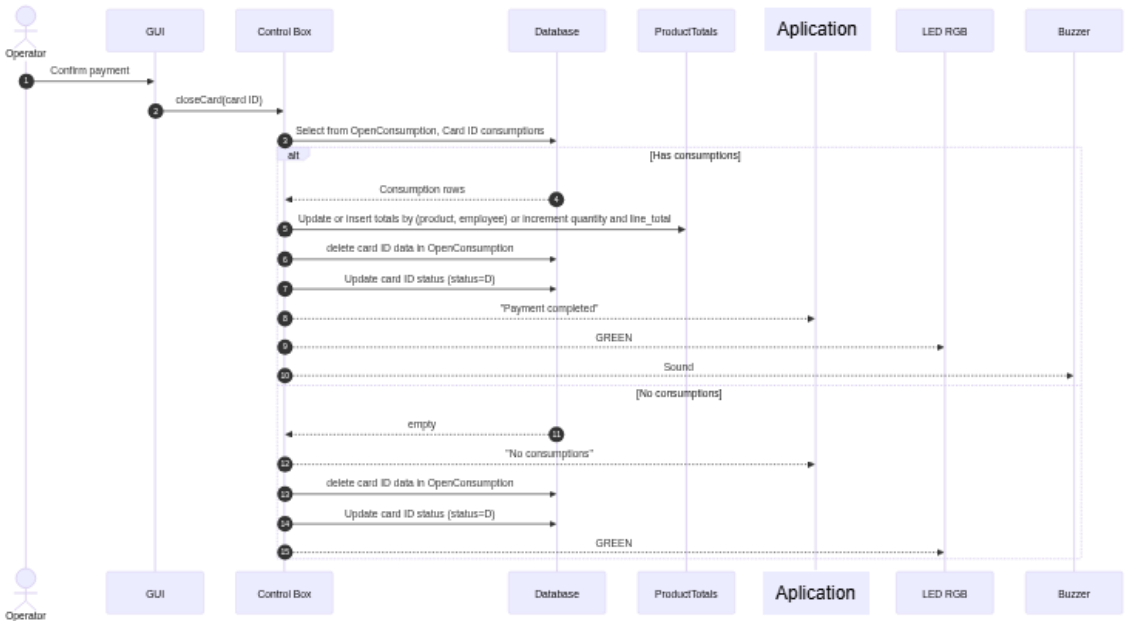

Figure 8 - Card closing Sequence diagram

3.4. Database

The NexiPass database supports the entire system cycle, from NFC card activation to consumption recording and closing, ensuring integrity, traceability, and privacy.

The structure was designed to simultaneously meet two distinct needs:

- **Operational**: where data is dynamic and reflects the real-time status of cards and transactions.
- **Analytical:** where consolidated data is stored for reports and statistics.

This division ensures a logical separation between sensitive (operational) and aggregated (analytical) data, optimizing performance and security.

3.4.1. Entities

The identified entities define the fundamental structure of the NexiPass database, representing the main elements involved in the system - from cards and products to users and consumption records.

Each of these entities has a specific and interdependent function within the system's operating cycle, ensuring the consistency and integrity of the stored information.

Thus, the model is composed of the following entities:

- **Users:** defines the system users and their respective roles (employee or manager)

- **Product:** contains the product catalog and the respective fixed unit price.
- **OpenCard**: represents the registered physical cards, containing status (A or C), telephone contact, and accumulated total.
- **OpenConsumption**: records all consumption lines associated with active cards.

3.4.2. Entity Relationships and Cardinalities

Relationships and cardinalities describe how entities in the NexiPass database interconnect with each other, ensuring referential coherence and consistency of stored data.

Each relationship defines the logical dependency between tables and the number of possible occurrences between them (1:1, 1:N), allowing us to represent how cards, products, consumptions, and users are associated in the actual functioning of the system.

Thus, the main relationships established are:

- OpenCard 1:N OpenConsumption — an active card can have multiple consumptions.
- Product 1:N OpenConsumption — a product can be recorded in multiple consumptions.
- Users 1:N OpenConsumption — each employee can record multiple consumptions.

3.4.3. Business Rules

The business rules establish the conditions and restrictions that determine the behavior of the database and the execution of its processes.

These rules ensure that all operations—from card activation to consumption entry and closure—are performed in a controlled manner, validating the states of entities and preventing logical inconsistencies in records.

Therefore, the following conditions apply:

- Consumption can only be recorded if card status = ‘A’.
- The total to pay field is automatically updated by the sum of the card's consumption lines.
- Card closure:
    - The consumption is aggregated by (product, employee) and increments the totals.
    - Consumption records are deleted.

- The card status is updated to ‘C’ and the phone field is cleared.
- The card becomes available again for another user.

- The phone number only exists during the ‘A’ status, ensuring privacy.

3.4.4. Visibility and Access

To ensure privacy, hierarchical access control, and logical data security, the following permission levels have been defined:

- Employee:
  - Has full read and write access to operational tables: OpenCard and OpenConsumption, except for sensitive fields (in this case, mobile phone numbers).
  - Can activate cards, record consumption, and close accounts, interacting directly with the operational database in real time.
  - Does not have access to the historic consumption analytical table, remaining limited to the operations of the active shift.
- Manager:
  - Does not intervene in real-time operations but has read access to the historic consumption analytical table, where aggregate sales totals are stored.
  - This view allows you to track the overall and individual performance of each employee, product, or terminal.
  - You do not need to edit operational data, ensuring the integrity of the analytical history.

3.4.5. Structure Benefits

The proposed architecture for the NexiPass database offers several advantages that are reflected in both operational performance and information reliability and security.

The separation between the operational domain and the analytical domain allows for efficient real-time data processing, while ensuring user privacy and operational integrity.

The main benefits include:

- Modular and scalable structure, allowing for future expansion or integration without compromising performance.
- Logical separation between operational and analytical data, optimizing processing and reducing query complexity.
- Secure reuse of physical cards, ensuring automatic deletion of personal data after each session is closed.
- Automatic and non-redundant updating of consumption totals, ensuring consistency between operational and analytical tables.
- Privacy and compliance with data protection principles by removing sensitive fields after payment.
- Immediate availability of performance reports, allowing managers to consult totals quickly and accurately by product, employee, or terminal.

3.5. Estimated Costs

The Table 2 below represents a brief analysis of the project costs.

Item	Quantity	Price (€)
Raspberry PI 4 Model B	1	55
RFID Reader	1	1,87
NFC Cards	10	2,29
Structure		~10
Other components		~15

Table 2 - estimated costs

4. Theoretical Foundation

4.1. Introduction to Embedded Systems

Embedded systems are specialized computer systems designed to perform dedicated functions within larger systems. Unlike generic computer systems, embedded systems are not generally programmable by end users and have applications defined at the project development stage. This specialization means that functionality is focused on capturing and analyzing data specific to the application domain.

The integration of multiple sensors and actuators highlights the importance of customized embedded solutions. These systems often operate on real-time operating systems (RTOS), which provide timing and resource management guarantees that are critical for critical applications.

4.1.1. System's Architecture

The architecture of embedded systems establishes the structural organization of their components and how they interact with each other. The hardware layer integrates sensors, actuators, and processing units, while the software layer is typically built on a customized real-time operating system, where it coordinates operations and implements control logic.

Modern embedded systems adopt a layered architecture:

- **Application Layer:** Business logic and data processing.
- **Middleware Layer:** Libraries and support services.
- **Kernel Layer (RTOS)**: Resource management, scheduling, and synchronization.
- **Hardware and Drivers Layer**: Direct interface with peripherals through device drivers.

4.1.2. Process and Thread management

4.1.3. Process

A process is an independent unit of execution that has its own resources and memory space. In embedded systems, the ability to execute multiple processes simultaneously is crucial, as different tasks need to be performed in parallel.

Processes generally have several states:

- **Running**: Actively using the CPU.
- **Ready**: Waiting for CPU allocation.
- **Blocked**: Waiting for I/O or synchronization.
- **Terminated**: Finished execution.

Understanding these states is essential to ensure the correct efficiency and real-time response of the system.

4.1.3.1 Inter-Process Comunication (IPC)

Inter-Process Communication (IPC) is essential for exchanging data between system components. The main mechanisms include:

- **Message Queues**: Enable asynchronous and decoupled communication between processes, ideal for distributed systems
- **Shared Memory**: Offers high-speed communication but requires robust synchronization mechanisms
- **Sockets**: Enable communication over a network or between local processes
- **Pipes**: FIFO structures for unidirectional communication

Message queues help prevent conflicts and ensure that data is processed in an orderly manner, crucial for systems where the sequence of events is critical.

4.1.4. Threads e Pthreads

Threads represent the smallest unit of a program that can be managed independently, allowing concurrent execution of tasks within a single process. The use of threads is particularly advantageous in embedded systems, as it allows for efficient use of resources and improves system response.

4.1.4.1 Pthreads (POSIX Threads)

The Pthreads library is widely used to implement multithreading in programs developed in C and C++. It offers a robust set of APIs that allow you to create, control, and synchronize multiple threads within an application. Because it is compatible with various operating systems, Pthreads stands out as a reliable solution, especially in applications geared toward embedded systems.

Among the main features offered by the Pthreads library, the following stand out:

- **Thread Creation**: Uses the pthread_create function to create threads
- **Thread Synchronization**: Mechanisms such as mutexes and conditional variables
- **Terminating Threads**: pthread_exit and pthread_join functions for controlled termination
- **Thread Attributes**: Stack configuration, scheduling policies, and properties

4.1.4.2 Task Scheduling Models

Task scheduling models define how the processor manages multiple activities, directly influencing the performance and responsiveness of computer systems.

There are three models:

- **Preemptive Model:** Among all tasks ready for execution, the task with the highest priority is always executed. Thus, lower priority tasks are preempted when a higher priority task becomes available, which serves to ensure immediate attention to critical operations.
- **Round-Robin Model:** Tasks are executed one at a time in a cyclical order. Each task runs until it completes or voluntarily yields the processor to another task. This model ensures that all tasks receive a fair share of CPU time.
- **Round-Robin with Time Slice Model:** Each task is allocated a fixed time slot for execution. If a task exceeds its time slice, it is suspended and moved to the end of the queue.

4.1.4.3 Context Switching

Context switching is an essential mechanism in multi-threaded programming, allowing the system to switch between threads efficiently. In embedded systems with real-time constraints, minimizing context switching overhead is critical to achieving low latency. This requires:

1. Save the current state of the thread (registers, program counter (PC))
2. Load the state of the next thread
3. Restore the context when the thread regains control

This operation has significant costs in terms of CPU cycles and must be carefully managed in real-time systems.

4.1.5. Synchronization and Mutual Exclusion Mechanisms

4.1.5.1 Mutexes and Condition Variables

Mutexes implement mutual exclusion, ensuring that only one thread accesses a critical section at a time.

Condition variables allow threads to wait for specific events and signal when a desired condition is satisfied, optimizing CPU usage.

4.1.5.2 Semaphores

Semaphores are more versatile synchronization primitives than mutexes, offering control over multiple resources.

Semaphores can be:

- **Binary semaphore**: Works like a mutex (values 0 or 1)
- **Counting semaphore**: Controls multiple instances of a resource

4.1.5.3 Priority Inversion and Priority Inheritance

Priority inversion is a critical problem in real-time systems, where a high-priority thread is blocked waiting for a low-priority thread. This problem is mitigated through priority inheritance, where the thread that acquires the mutex temporarily inherits the priority of the blocked thread.

4.1.6. Thread Security

Thread safety is crucial when dealing with shared resources in a multi-threaded environment. To achieve this, conditions such as the following must be ensured:

- Atomicity: Operations that cannot be interrupted
- Visibility: Ensure that modifications are visible to all threads
- Ordering: Maintain the appropriate sequence of operations
- Data segregation: Minimize shared data

4.1.7. Real time systems

Real-time systems depend not only on the logical result, but also on the timing of the results. There are two types:

- Hard Real-Time: Failure to meet a deadline is catastrophic.
- Soft Real-Time: Failure is tolerated occasionally.

4.1.8. I/O System

The I/O system forms a critical layer in embedded systems, integrating devices, drivers, subsystems, and the operating system kernel. It provides a standardized approach to accessing input/output devices, enabling data flow between sensors, the CPU, and the application layer.

4.1.8.1 Device Drivers

Device drivers are fundamental components that reside within the operating system kernel and operate at a lower level than the application architecture. This design allows for efficient communication with the hardware, handling data transmission with minimal latency.

Device drivers are categorized into two main types:

- Character Device Drivers: Handle data one character at a time, suitable for event-driven devices (keyboards, serial ports). They feature direct open, close, read, and write operations.

- Block Device Drivers: Transfer data in large fixed-size blocks, more efficient for high-volume data storage and retrieval (hard disks). They require cache and buffer management.

Wireless Communication in Embedded Systems

Wireless communication technologies are a fundamental pillar of contemporary embedded systems, particularly in applications that require network connectivity and remote data access. The integration of wireless communication protocols, such as Wi-Fi, into embedded systems enables real-time data transmission, synchronization with remote servers, and centralized device management.

Wi-Fi is a wireless communication protocol based on the IEEE 802.11 standard, which provides connectivity in wireless local area networks (WLANs). In embedded systems, Wi-Fi offers significant advantages in terms of range, transmission speed, and compatibility with existing network infrastructures.

4.1.8.2 Communication Models

- Half-Duplex: Bidirectional communication, but not simultaneous.
- Full-Duplex: Simultaneous bidirectional communication.

4.1.8.3 Características Críticas

- Latency: Time elapsed from transmission to reception.
- Bandwidth: Maximum data transfer rate.
- Reliability: Ability to communicate without errors.
- Robustness: Resistance to interference and environmental conditions.

4.1.8.4 TCP/IP Protocol Stack over WiFi

In embedded systems, WiFi communication is typically used with the TCP/IP protocol stack:

- Application Layer: HTTP/HTTPS, MQTT, CoAP.

- Transport Layer: TCP (reliable) or UDP (unreliable).
- Internet Layer: IPv4 or IPv6
- Link Layer: WiFi (IEEE 802.11)

This architecture allows the system to communicate with remote servers, cloud systems, and other devices in a standardized and interoperable manner.

4.1.9. I²C communication Protocol

The I²C (Inter-Integrated Circuit) bus is a multi-master, multi-slave communication protocol designed for low-speed, medium-distance data transmission. It is particularly useful in embedded systems for efficient, synchronized communication with multiple peripheral devices.

The I²C bus operates with only two main lines:

- SDA (Serial Data): Transfers data between devices, supporting bidirectional operations
- SCL (Serial Clock): Generates the clock signal that synchronizes the transfer

Each device has a unique address, and the simplicity of having only two lines reduces the number of wires required and the GPIO pins used, making I²C ideal for managing multiple devices.

SPI (Serial Peripheral Interface) communication

SPI is a synchronous, full-duplex communication protocol that allows simultaneous data transfer between a master device and one or more slave devices. Unlike the I²C protocol, which operates at lower speeds, SPI offers high transmission rates, making it ideal for applications that require fast, real-time communication with peripherals such as RFID readers, high-speed sensors, and memory modules. It should include:

- Technical characteristics: Synchronous protocol, full-duplex, master-slave.
- Communication lines: MOSI, MISO, SCK, CS.
- Advantages: Low latency, high transmission rate.

- Comparison with I²C: SPI is faster for applications that require real time.

Applications in embedded systems: Reading sensors, memory modules, communication with peripherals**.**

4.1.10. Message Queues

Message queues provide a robust mechanism for communication between threads, ensuring that messages are processed in FIFO order, which is essential for applications where the sequence of events is critical.

Implementations such as POSIX message queues offer: Message prioritization; Timeouts on blocking operations; Reliable synchronization.

4.1.11. SQL and Relational Databases

Relational databases are the most widely used model for storing and managing structured information.

They are based on organizing data into tables (entities) composed of columns (attributes) and rows (records), where relationships between tables are defined using primary keys and foreign keys.

This model ensures referential integrity, elimination of redundancies, and data consistency, which are essential characteristics in critical and distributed systems.

SQL (Structured Query Language) is the standard language for defining and manipulating data in relational databases.

With SQL, it is possible to create and modify schemas (Data Definition Language – DDL), insert, update, and delete data (Data Manipulation Language – DML), and query specific information using SELECT statements, thus ensuring flexibility and accuracy in data access.

4.1.11.1 Primary Keys

A Primary Key (PK) is a unique identifier assigned to each record within a table.

Its main purpose is to ensure that each row is distinct and can be referenced unambiguously, allowing the database to distinguish between different entities.

In technical terms, a column (or set of columns) defined as a Primary Key cannot contain duplicate or null values.

4.1.11.2 Foreign Keys

A foreign key (FK) establishes a relationship between two tables, creating a logical link between their data.

An FK in one table points to the PK of another, ensuring that the referenced values exist in the target table — this mechanism is called referential integrity.

4.1.12. Enum Class

The enum class type, represents a finite set of named values used to describe states or categories in a symbolic and readable way.

Unlike traditional enumerations, enum class has two key advantages:

- Type safety — values belong to their own type and are not implicitly converted to integers, preventing comparison errors or accidental assignment.
- Controlled scope — identifiers defined in enum class do not pollute the global namespace and are accessed through the scope resolution operator (::).

5. Design phase

Once the analysis phase has been completed, during which all system requirements and restrictions are defined, the design phase follows, which is essential to ensure a more organized and fluid implementation later. It is important that this phase is left in detail to facilitate the work of future teams that will carry out the implementation. To this end, it is essential that the design phase is concise, accurate, and clear to simplify future development since errors at this stage may require the system to be redesigned and reimplemented.

5.1. System Analysis Review

To ensure that everything remains consistent with the requirements, restrictions, and objectives of the System described in the analysis phase, a review of the analysis phase must be performed.

The NexiPass system was designed using tools such as Buildroot to generate customized Linux images for the Raspberry Pi, and uses PThreads for efficient management of multiple threads, enabling robust and responsive real-time operation. It focuses on monitoring and recording events related to NFC card readings, consumption management, and transactions, ensuring data operability and integrity through procedures defined for each system event. The device integrates an application for immediate visualization and adjustment of operations by the operator. The system architecture is divided into three layers: the Linux operating system layer, responsible for direct interaction with physical devices (Wi-Fi, RFID readers, buzzer); the middleware, which provides abstraction and multithreaded data management; and the application, which concentrates the user logic and the graphical interface developed in Flutter. The database supports synchronous and asynchronous operations, dividing operational and analytical data to optimize performance and segregation of sensitive information. The system adopts automatic privacy policies, where, for example, the mobile phone number is removed from the database as soon as the card transitions to an inactive state, thus ensuring the protection of the user's personal data and compliance with good security practices. The entire operational flow, from card activation, consumption recording, payment confirmation, and status transition, is designed to be efficient and scalable to other consumption and access environments.

5.2. Hardware Specification

In embedded systems, hardware components are essential for implementing the desired functionalities. To do this, it is necessary to study the different hardware components of the system to

understand how they work and how they interconnect, from sensor modules to actuators. These components play a key role in data acquisition, processing, and transmission, as well as in interaction with the outside world. In addition, interfaces facilitate communication between the various components, ensuring continuous data flow and real-time responsiveness. Understanding how the different hardware elements work is essential to ensure a robust, efficient, and logically consistent system with the desired characteristics, always meeting the defined requirements and constraints.

5.2.1. Raspberry Pi 4 model B

The Raspberry Pi 4B acts as the central control unit of the system; it is therefore the most important element of the system, responsible for controlling and interacting with the various sensors and actuators.

5.2.1.1 Specifications

- **Processor**: Broadcom BCM2711, Quad-Core Cortex-A72 (ARM v8) 64-bit SoC @1.5GHZ
- **RAM memory**: 2GB LPDDR4-3200 SDRAM**.**

5.2.1.2 Connectivity

- **Ethernet**: Gigabit Ethernet (RJ45)
- **Wi-Fi**: Dual-band 2.4/5.0 GHz (802.11ac)
- **Bluetooth**: 5.0 /BLE

5.2.1.3 Expansion Ports and USB

- **USB**: 2 x USB 3.0; 2 x USB 2.0.
- **GPIO**: 40-pin GPIO header; Voltage: 3.3V; Max. current per pin: 16mA; various alternative functions (I2C, SPI, UART, PWM...) Figure *10*
- **MIPI DSI/CSI**: For display and camera

5.2.1.4 Video and Audio Output

- 2 x micro-HDMI (4Kp60)
- **Audio/Video**: 4-pin stereo + composite video

- SD Memory Card Slot for microSD card for OS and Data
- **Power supply**
    - **USB-C**: 5V DC (minimum 3A)
    - **GPIO header**: 5V DC
    - **Power over Ethernet**: Requires optional PoE HAT.
    - **Recommended consumption**: Min. 3A
- **Operating Temperature**: 0°C to 50°C.

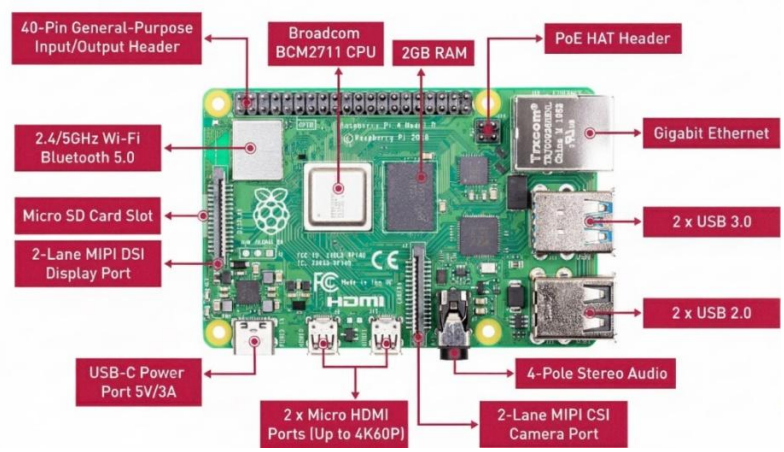

Figure 9 - Raspberry Pi 4 Model B Components

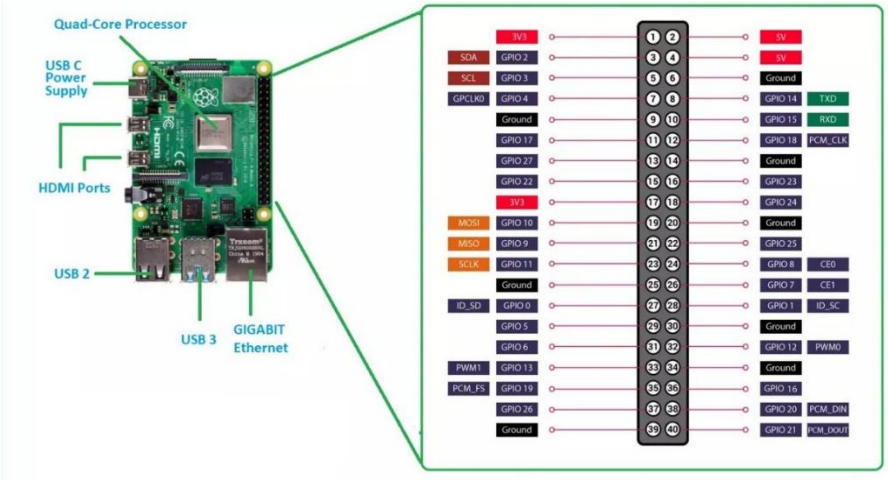

Figure *10 -Raspberry Pi 4 GPIO Pinout Diagram*

5.2.2. Buzzer

To provide feedback to the user in the event of unexpected scenarios during use of the system, a buzzer has been implemented to alert users to unpaid cards, unsuccessful activation, or activation that has not been performed.

5.2.2.1 Specifications

- **Type:** Piezoelectric
- **Sound pressure level**: 70 dBA
- **Tone**: Continuous
- **Frequency**: 4 kHz
- **Voltage rating**: 3 V
- **Termination style**: Solder Pin
- **Mounting type**: Through Hole

Figure 11 - Active Buzzer Module

5.2.3. LED RGB

The RGB LED is a tricolor light-emitting diode that combines three emitters (red, green, and blue) in the same package, allowing any color to be generated through additive mixing. On the Raspberry Pi, it is controlled by the GPIO pins (3.3 V) using digital or PWM outputs. It is a Common anode LED.

Figure 12 - Common anode RGB LED

5.2.3.1 Electrical Specifications

Parameter	Red	Green	Blue
Direct voltage (Vf)	1.8–2.4 V	2.8–3.6 V	2.8–3.6 V
Typical current (If)	10–20 mA	10–20 mA	10–20 mA
GPIO voltage (máx.)	3.3 V	3.3 V	3.3 V
GPIO Current (máx.)	16 mA	16 mA	16 mA

Table 3 - RGB LED Specifications Table

5.2.4. MODULE RFID-RC522

The MFRC522 module is widely used for reading/writing RFID/NFC cards. This module has three modes of communication with the board: SPI, I2C, and UART, each with its own advantages and disadvantages. It is the component responsible for operating as an interface between the antenna (which communicates with the card) and the Raspberry Pi, providing data such as the UID (unique card identifier), card memory data, and writing data to that same memory.

5.2.4.1 PINOUT

- SDA/SS (Chip Select): SPI device selection
- SCK (Clock SPI)
- MOSI (Master Out Slave In, SPI)
- MISO (Master In Slave Out, SPI)
- IRQ: Interruption (optional)
- GND: Ground
- RST: Reset
- VCC: 3.3V

5.2.4.2 Typical consumption: 13–26 mA

5.2.4.3 Supported interfaces:

- SPI até 10 Mbit/s
- I²C até 400 kbit/s
- UART 115200 bps

- This module is composed of the following internal blocks:
  - Analog Interface: Handles radio frequency communication between the module and the antenna (card/contactless).
  - Contactless UART: Decodes signals between antenna and digital.
  - FIFO Buffer: Temporarily stores read/written data.
  - Register Bank: Stores settings and parameters.

- Serial UART/SPI/I²C Bus: Communicates digitally with the Raspberry Pi.

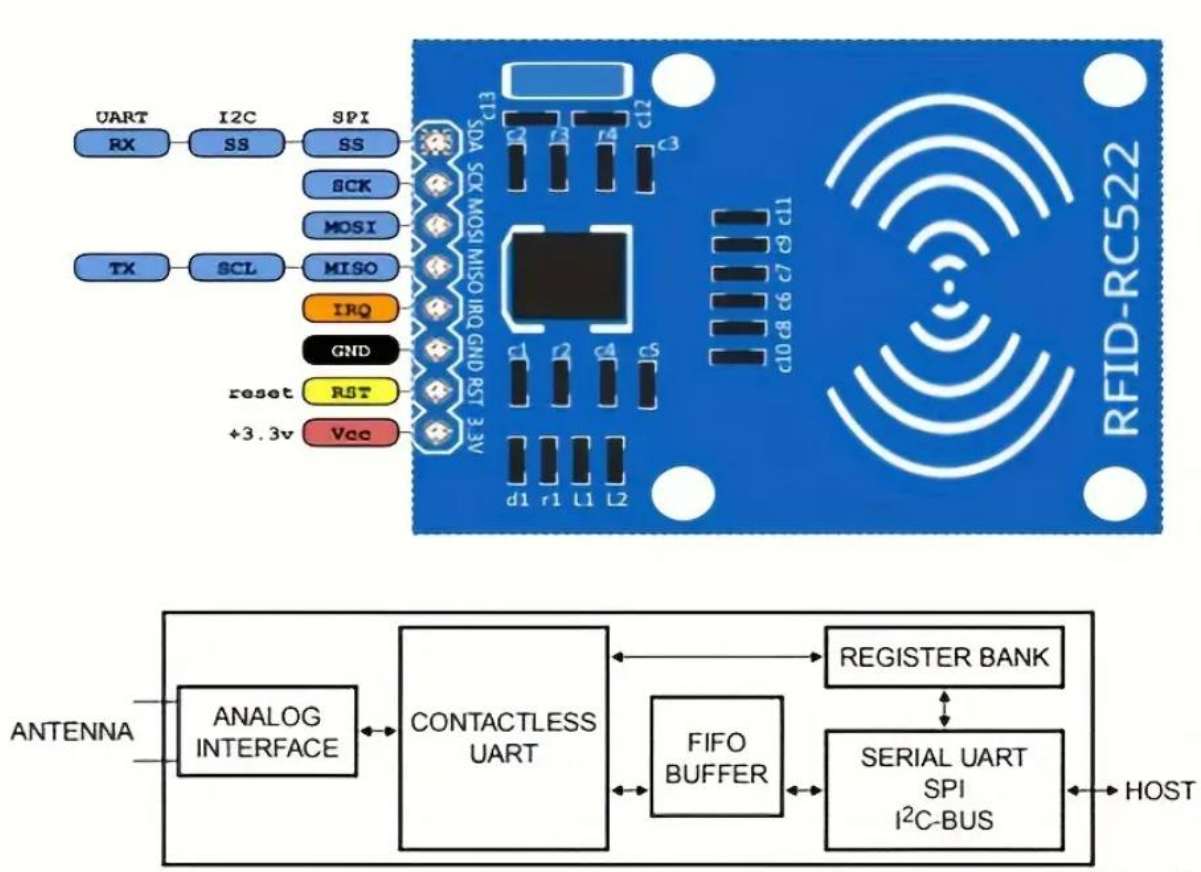

Figure 13 - RFID-RC522 Module Pinout and Block Diagram

5.2.5. SD Card

An SD card was used to store the Raspberry Pi 4 software. This card has 32 GB, which is sufficient for our implementation.

Figure 14 - Kingston 32GB microSD Card

5.2.6. Power Supply

A USB-C power adapter connected directly to the mains is used to supply power to the system's main board. This adapter is suitable for powering devices with a Type-C connection. This adapter meets the requirements for the correct power supply to the Raspberry Pi board, as it accepts input voltages between 100-240V at 50/60Hz, which is sufficient to support the electrical networks of different countries. Its output provides a stable output voltage of 5V and up to 3A, i.e., it provides total power corresponding to 15W, sufficient for the correct and reliable operation of the board to be used (Raspberry Pi).

- **Input**: 100-240V, 50/60 Hz (AC)
- **Output:** 5V (DC)
- **Max current**: 3mA
- **Output connector:** Usb type-c

Figure 15 - Power Supply

5.2.7. SPI

In this project, SPI communication allows the Raspberry Pi 4 B to exchange data with the main peripheral: the RFID-RC522 module.

This communication protocol is synchronous and full duplex, allowing data to be transmitted and received simultaneously between the master device and slave devices.

The Raspberry Pi acts as the master on the bus, responsible for generating the clock signal (SCLK) and controlling communication through the MOSI (Master Out, Slave In), MISO (Master In, Slave Out), SCLK (Serial Clock), and CS (Chip Select) lines.

Each device is selected individually through a dedicated CS pin, which, when active, allows exclusive communication between the Raspberry Pi and the peripheral in question.

During the communication process, the master sends the data bit by bit through the MOSI line, while simultaneously reading the responses sent by the slave through the MISO line.

This exchange occurs in sync with the clock signal (SCLK), ensuring accurate and stable data transfer.

The SPI protocol is widely used in embedded systems due to its low latency, high transmission rate, and simplicity of implementation.

In the context of this project, the RFID-RC522 module is responsible for reading and authenticating RFID cards.

The use of the SPI bus ensures fast and reliable communication between components, making it ideal for applications that require real-time response and high data integrity.

5.3. Hardware Connections

At this stage, it is essential to specify the connections between the different parts of the design. It is important to explicitly state all physical hardware connections, so that they can be used later in the implementation phase.

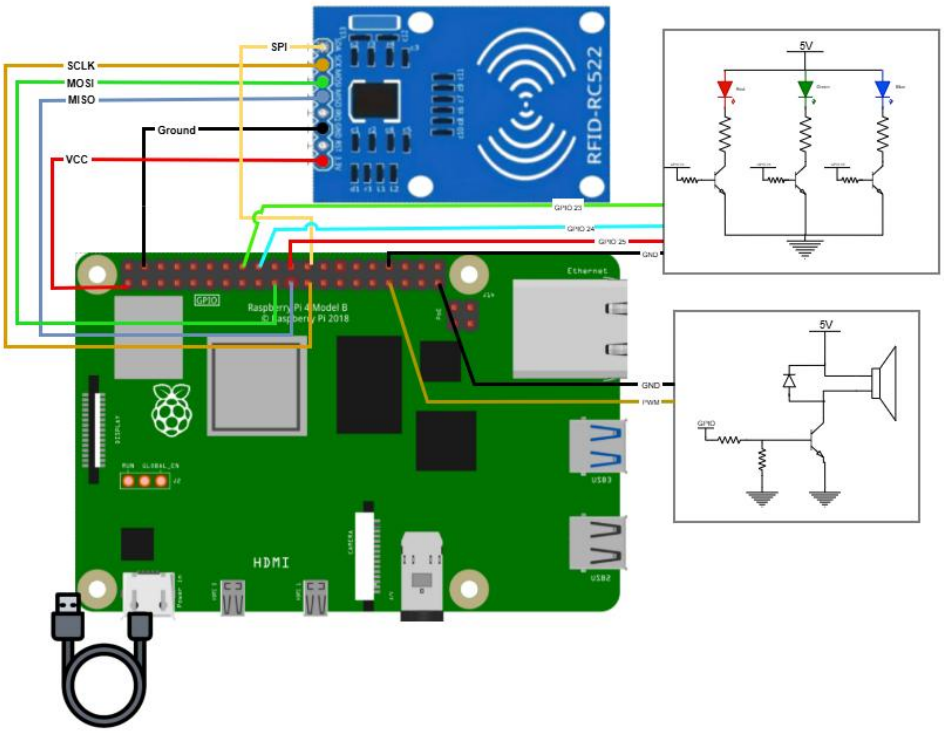

Figure 16 - Hardware connections

5.3.1. GPIO Specification

GPIO	Description
GPIO 23	Output pin to turn on/off red rgb light
GPIO 24	Output pin to turn on/off green rgb light
GPIO 25	Output pin to turn on/off blue rgb light
GPIO 9	MISO pin for SPI communication with RFID sensor module
GPIO 10	MOSI pin for SPI communication with RFID sensor module
GPIO 11	SCLK pin for SPI communication with RFID sensor module
GPIO 8	CE0 for SPI communication with RFID sensor module
GPIO13	PWM Output for buzzer control(pwmchip0/pwm1)

Table 4 - GPIO Specification

5.4. Software Specification

The software for this project will be modular, object-oriented (OOP), and multithreaded, designed to support real-time execution.

The modules in this system will have specific functions such as receiving a request for an action, processing that action, and reading and storing data in the database.

5.4.1. Class Diagram

Since this is a project that follows OOP (Object Oriented Programming) implementation, it was necessary to create a class diagram to understand how the classes interact with each other.

This system was designed according to a modular architecture, consisting of three main processes:

- **CONTROL_BOX_PROCESS**: This process represents the logical core of the system, and is therefore the main process, running on the Raspberry Pi board. It is responsible for all internal coordination (system logic), communication between peripherals, and data management, integrating classes such as ApiController, CardService, NexiPass Database, and FeedbackController.

- **USER_INTERFACE_PROCESS:** This process corresponds to the mobile application, which functions as a GUI for users. The APP communicates with the ApiController class by sending and receiving commands and responses in real time. This communication is done via a wireless connection (Wi-Fi). This process ensures direct interaction with the user, allowing access to login, card activation, consumption recording, and viewing of totals.

- **HARDWARE_INTERFACE_PROCESS:** It is responsible for grouping the physical elements of the system. The NFC Reader is the RFID transmitter module that reads the unique identifier (UID) of each card and sends the information to the ApiController, allowing the card to be associated with the user. The FeedbackController controls the LED and Buzzer to generate appropriate signals depending on the operations.

This division is used to ensure a clear separation between the control, interface, and physical interaction layers of the system, thereby enabling compliance with parameters such as greater scalability, easy maintenance, and data security.

Figure 17 shows the connections between the different parts of the processes. There are different types of interactions. Within the CONTROL_BOX_PROCESS block, the different parts

interact with each other through aggregation, which means that one class is part of another, but can exist independently of the whole.

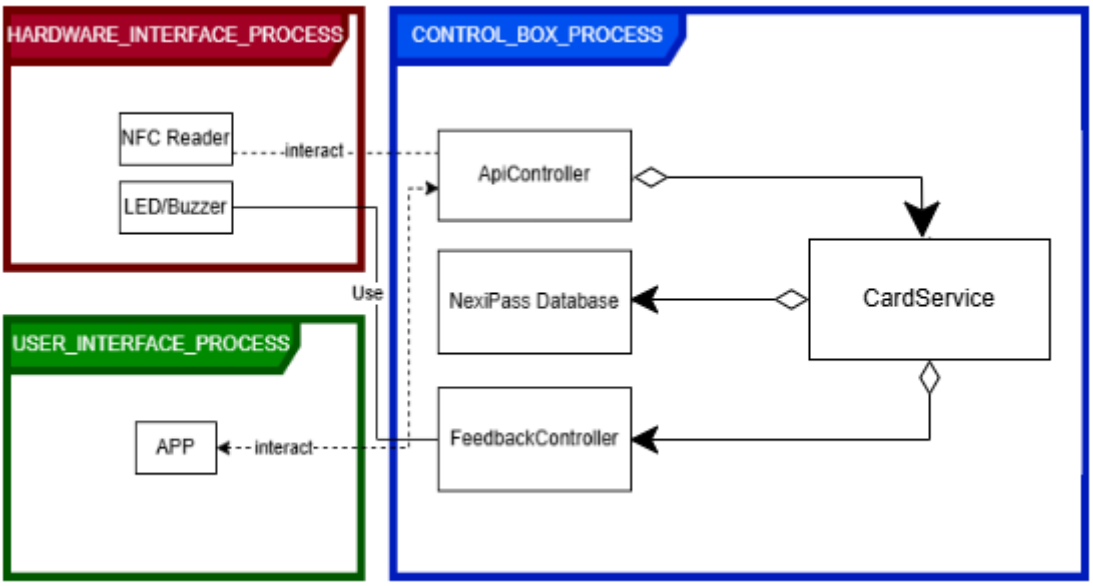

Figure 17 - System Architecture Diagram

5.4.2. Database description

The following Figure *18* shows the relational model of the NexiPass database, which structures the main entities of the consumption control and card management system.

The USERS entity stores information about system users, such as username, password, and role. The primary key is the user_id used as a reference in other tables, ensuring the unique identification of each employee.

The PRODUCT table contains the list of available products and is referenced in OPENCONSUMPTION and PRODUCTTOTALS.

The OPENCARD entity represents the active cards in the system. Each card is identified by a primary key (card_id) and contains the associated phone number and the card status, which can be either Active or Deactive.

The OPENCONSUMPTION table constitutes the transactional core of the database, recording each consumption operation performed. Each row contains direct references, foreign keys, to the OPENCARD, PRODUCT, and USERS tables, through the card_id, product_id, and employee_id fields, respectively. This structure ensures complete traceability of each

consumption, from the card used to the product sold and the employee responsible for the operation.

Finally, the PRODUCTTOTALS table statistically aggregates consumption results by product and employee, this being an analytical table. Its primary key is composed of the fields (product_id, employee_id), reflecting the unique combination between product and employee.

The model ensures consistency between entities, avoiding redundancies and ensuring data integrity through Primary Keys (unique identifiers) and Foreign Keys (referential links between tables). This structure allows for fast and consistent queries, which are essential for the real-time operation of the NexiPass system.

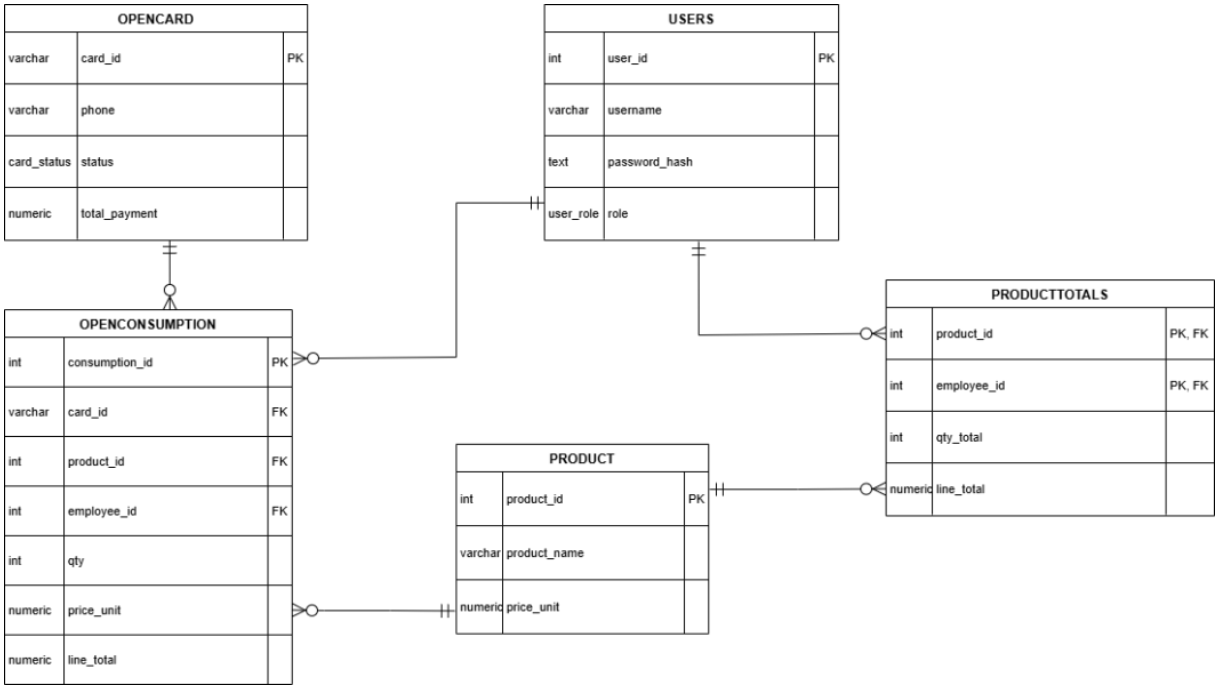

Figure 18 - Database Entity-Relationship Diagram

5.4.3. Structs

During the design phase of this system, several structs were defined that act as Data Transfer Objects (DTOs), i.e., structures used to transport data between different system modules. These structs contain only the fields strictly necessary to represent the data manipulated by the Database class functions and consumed by the CardService and ApiController classes.

This model ensures clear and secure information transfer between threads, i.e., each struct is produced by copying from the database, avoiding synchronization and shared memory issues.

In other words, the database temporarily stores the current data, while the structs are secure working copies, meaning that only when changes are necessary is the respective write/update operation performed in the Database with the data already processed and stored in the structs.

Thus, the system preserves a strict separation between data, logic, and concurrency, ensuring scalability, maintainability, and consistency in internal communication.

The defined structs (listed below) correspond to the logical entities of the relational database model, representing in a structured way the operations of user authentication, product listing, consumption recording, and consultation of totals and aggregate summaries.

5.4.3.1 UserDTO

The UserDTO struct represents the essential data of an authenticated user. It contains the fields user_id, username, and role, corresponding directly to the users table in the database.

It is used in the authentication process, allowing the system to identify whether the user is an owner (manager) or an employee.The object is created by the authenticateUser() function of the Database class and sent to the service layer (CardService).

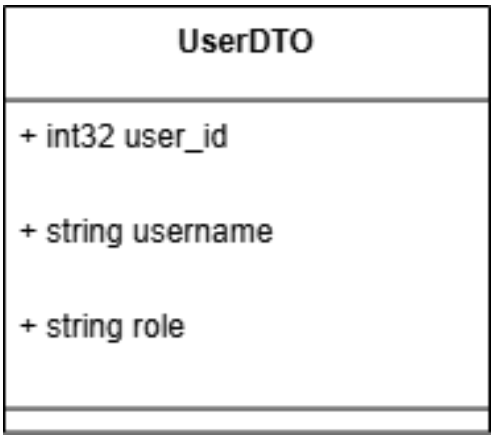

Figure 19 - UML Struct Diagram – UserDTO

5.4.3.2 Product DTO

The ProductDTO struct describes the products available for consumption, with the fields product_id, name, and price_unit. It reflects the product table and is used to display product lists in the mobile application and during consumption registration.

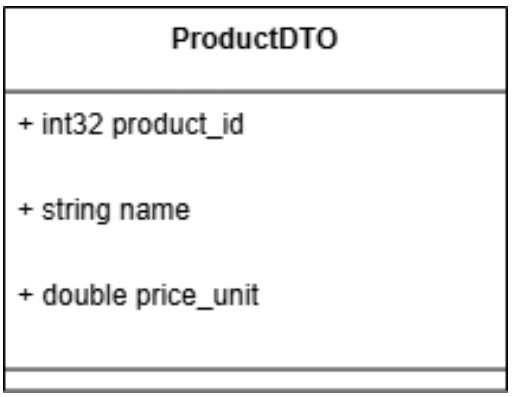

Figure 20 - UML Struct Diagram – ProductDTO

5.4.3.3 ConsumptionLineDTO

The ConsumptionLineDTO struct represents an individual consumption line associated with an active card.

Each instance includes information about the product consumed, the quantity, the unit price, and the total for the line.

These structures are grouped into arrays within a CardSummaryDTO, forming the complete list of a user's consumption.

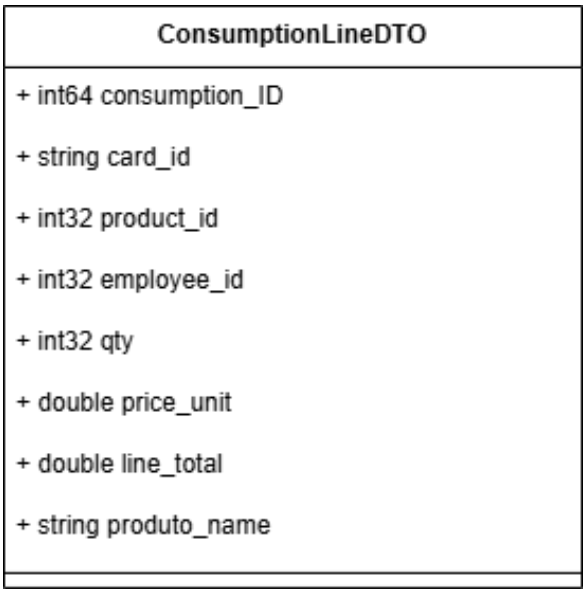

Figure 21 - UML Struct Diagram – ConsumptionLineDTO

5.4.3.4 CardSummaryDTO

The CardSummaryDTO struct aggregates the summary of a card's status and contains a vector of ConsumptionLineDTO, whose content is read-only.

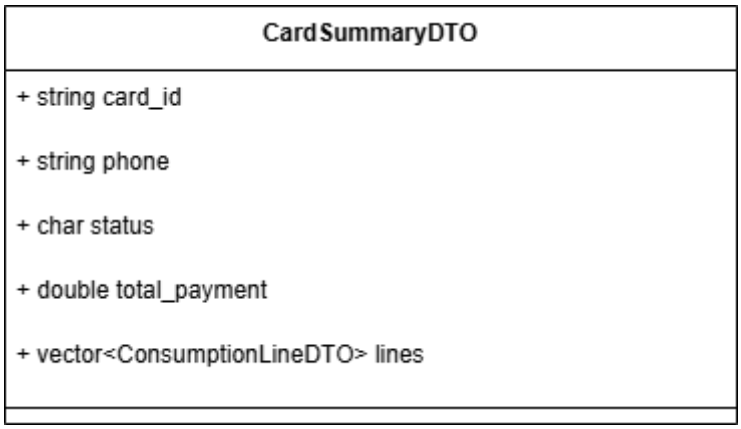

Figure 22 - UML Struct Diagram – CardSummaryDTO

5.4.3.5 TotalsRowDTO

The TotalsRowDTO struct is used to store analytical results (for the owner only), representing sales totals by product and employee.

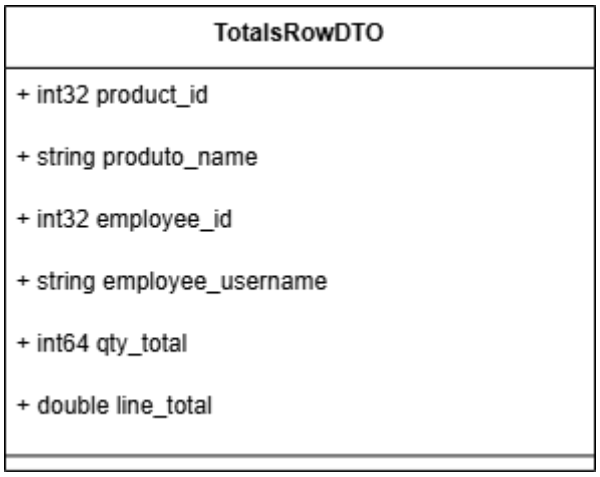

Figure 23 - UML Struct Diagram – TotalsRowDTO

5.4.4. Classes

5.4.4.1 DbResult

The DbResult enum class defines the return status of operations performed by the Database class.

Each value represents the result of an action on the database, allowing for structured error handling and clear communication between modules, including values such as:

- **Ok** – operation completed successfully.
- **NotFound** – record does not exist.
- **InvalidState** – operation invalid for the current state.
- **Error** – generic execution error.
- **ConnectionLost** – database connection failure.

This approach promotes uniform handling of exceptions and error states, avoiding the direct use of exceptions and simplifying the detection and propagation of failures in interactions between business logic and the data layer.

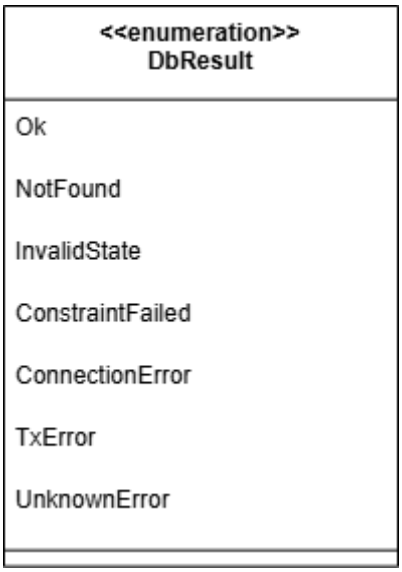

Figure 24 - UML Enum Class Diagram – DbResult

5.4.4.2 Cardservice

The CardService class implements the system's core business logic.

It is responsible for coordinating operations related to user cards, encapsulate calls to the Database class and the FeedbackController.

In this way, it acts as an intermediate layer between the database and the other system modules.

Its main functions include:

- **activateCard()** – activates an NFC card by associating it with the user's phone number.

- **addConsumption()** – records a new consumption, updating the internal status of the card and notifying the feedback interface.
- **deactivateCard()** – deactivates a card after checkout, ensuring the consumption cycle is closed.
- **getCardSummary()** – queries and returns the status of an active card.
- **getProductList()** – provides the list of products for the interface module.

CardService has direct dependencies on the Database and FeedbackController classes, and these relationships are of the aggregation type, as its functionality depends entirely on them.

Thus, this class constitutes the functional core of the system and ensures consistent execution of operations between threads and modules.

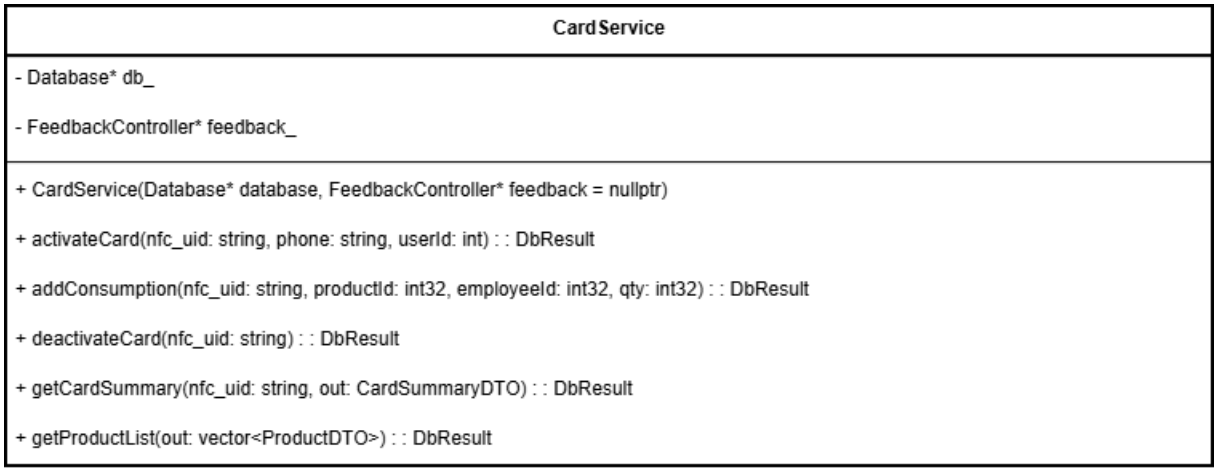

Figure 25 - UML Class Diagram – CardService

5.4.4.3 Database

The Database class is responsible for the data access layer and direct communication with the PostgreSQL database.

It contains all the functions necessary to manage CRUD (Create, Read, Update, Delete) operations, including user authentication, card activation and deactivation, consumption recording, and summary and total queries.

Through the defined methods, this class abstracts the complexity of interacting with the database, exposing a simple and secure interface for the upper layers.

Among its main methods are:

- **authenticateUser()** – validates user credentials and returns the respective data (UserDTO);0
- **activateCard()** and **closeCard()** – manage the lifecycle of active cards;
- **registerConsumption()** – records the consumption of a product by a specific employee;
- **getCardSummary()** and **getTotals()** – produce summaries and reports based on stored data;
- **listProducts()** – returns the list of products available for selection in the system.

The class maintains internal attributes such as the connection string (connString_), a generic pointer to the connection (conn_), and the restricted view mode for employees (employeeViewMode_), ensuring access control and data privacy.

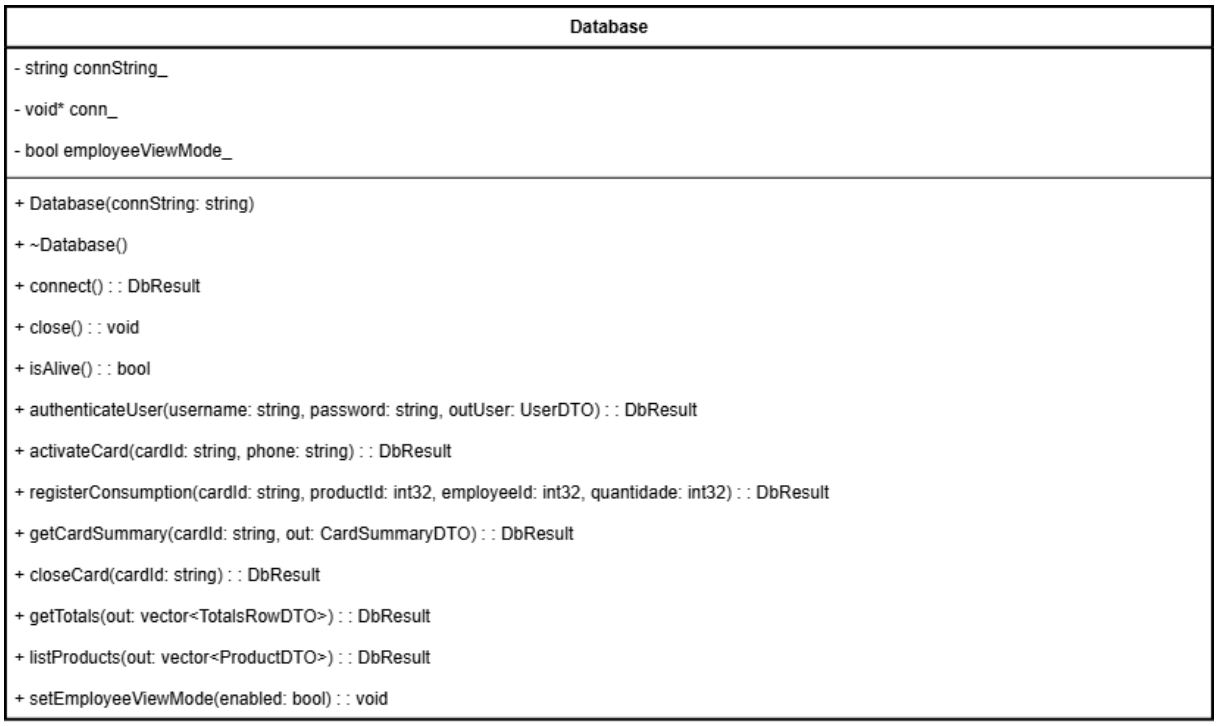

Figure 26 - UML Class Diagram – Database

5.4.4.4 Feedback controller

The main function of the FeedbackController class is to manage the system's visual and audible feedback, controlling the LEDs and buzzer on the Control Box.

It is called by CardService whenever there is a relevant change in the system status, such as card activation, authentication error, or checkout completion.

Its methods directly reflect the physical states of the hardware interface:

- **activateFB()** – signals the activation of a card with light and/or sound feedback.
- **deactivateFB()** – deactivates active feedback after use.
- **errorFB()** – emits an error signal, useful for NFC reading or authentication failures.
- **checkoutFB()** – signals the end of the consumption process.

Through this class, the system ensures intuitive communication with the end user, translating the logical states of the software into perceptible physical signals.

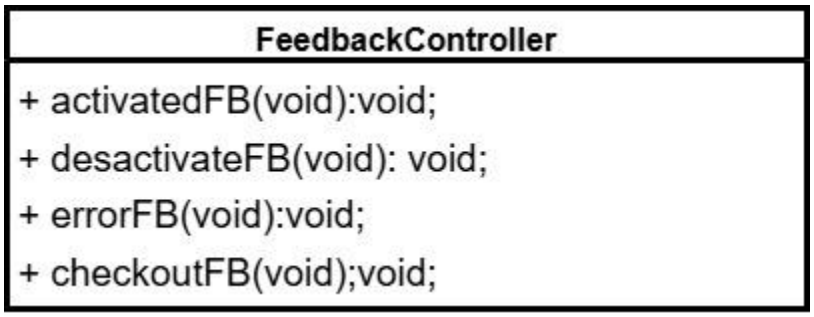

Figure 27 - UML Class Diagram – FeedbackController

5.4.4.5 ApiController

The ApiController class acts as the central coordination point of the system, responsible for receiving requests from the NFC reader and the mobile application, forwarding them for appropriate processing.

It generates three internal threads: **thNFC_**, **thNetwork_**, and **thWorker_** . which implement the producer/consumer model. The first two capture external events and place them in a protected queue, while the third processes these events synchronously through CardService, communicating the results with the Database and FeedbackController.

In this way, ApiController ensures that all operations are performed concurrently, securely, and in an orderly manner, ensuring predictable and consistent response times with a real-time system.

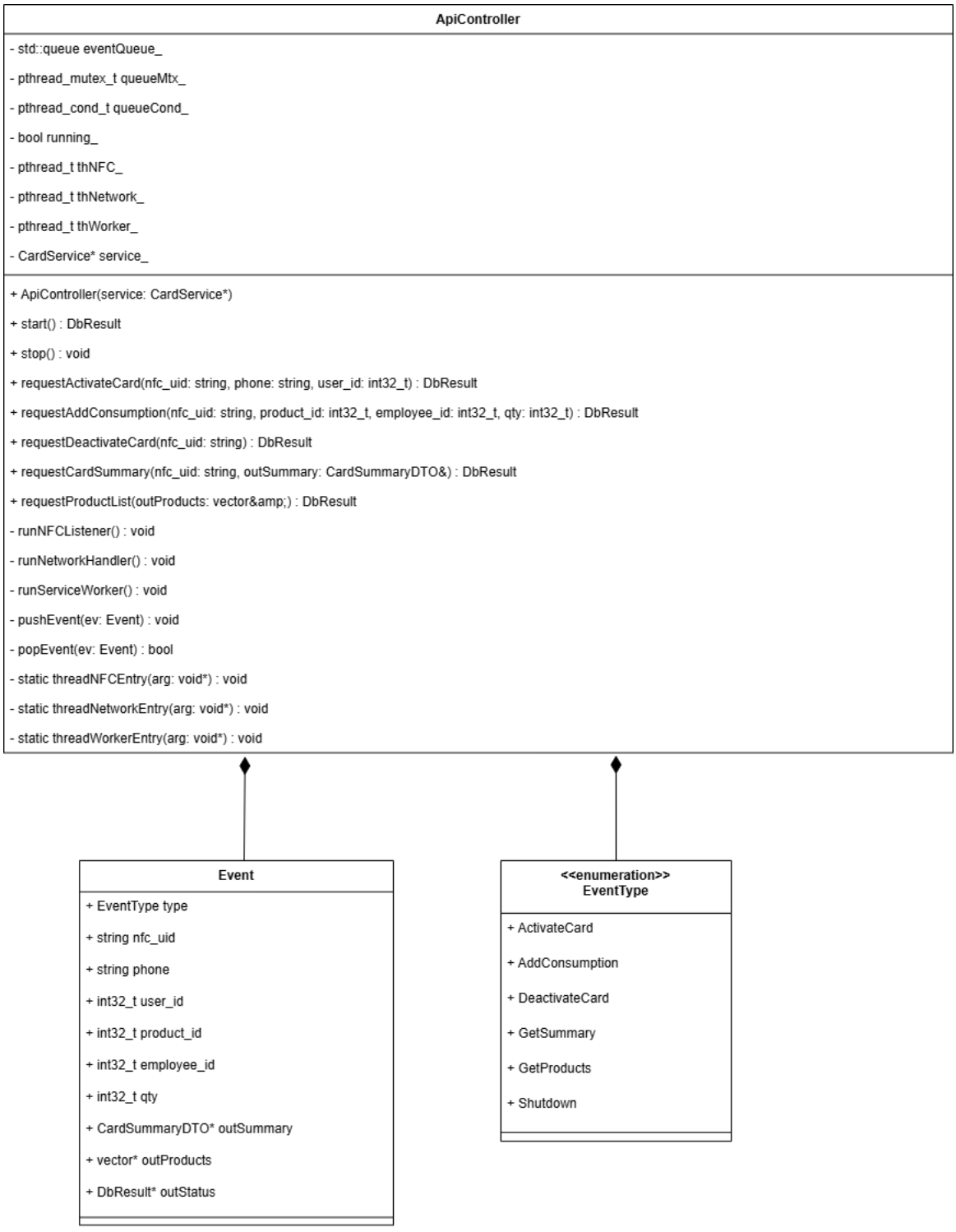

Figure 28 - UML Class Diagram – ApiController, Event, and EventType

5.5. Classes flowcharts

5.5.1. Database flowcharts

The following figures illustrate how the Database class functions were designed, from the communication lifecycle with the PostgreSQL database, which are the DbResult connect() (Figure 31), close() (Figure 30), and isAlive() (Figure 33) functions, to the functions that represent the main business actions, which are:

DbResult authenticateUser(...) (Figure 29): Verify if there is the user selected on Database.

, DbResult activateCard(…)(Figure 34): Serves to put card's status 'A', meaning that the card becomes activated.

DbResult registerConsumption(...) (Figure 32): Used to add consumptions to card, if it is activated.

DbResult getCardSummary(...) (Figure 35): To view every consumption that is already on the card.

DbResult closeCard(...) (Figure *36*): Close the card, returning the total, erasing all data on that and putting the status desactivated. Doing this we ensure that the card can be used for other user.

DbResult getTotals(...) (Figure *37*): Used to get card's total amount.

DbResult listProducts(...) (Figure *38*): To view every product that exists on database.

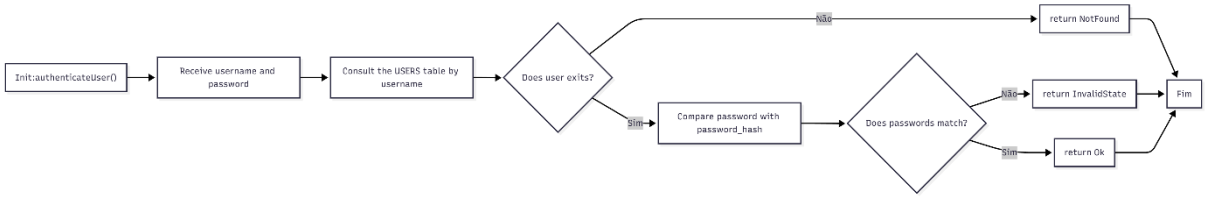

Figure 29 - authenticateUser() Function Flowchart

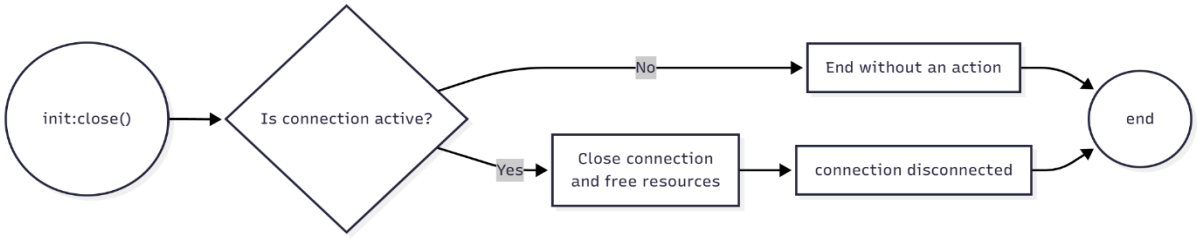

Figure 30 - close() Function Flowchart

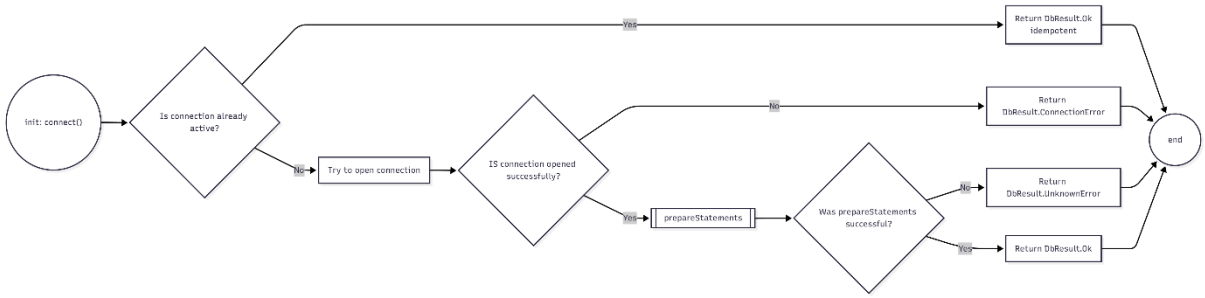

Figure 31 - connect() Function Flowchart

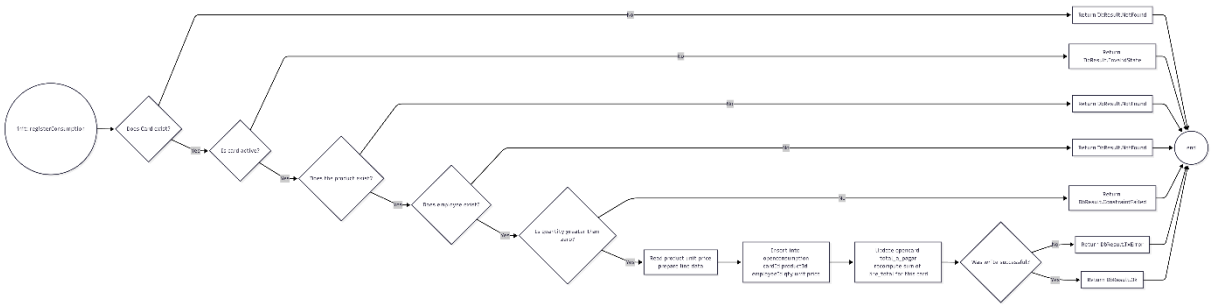

Figure 32 - registerConsumption() Function Flowchart

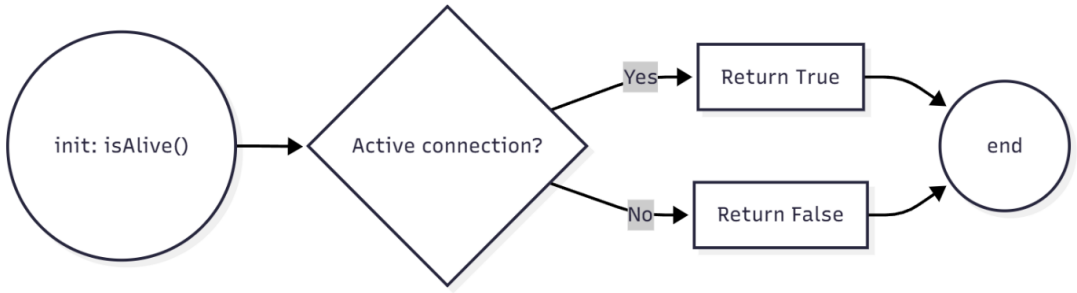

Figure 33 - isAlive() Function Flowchart

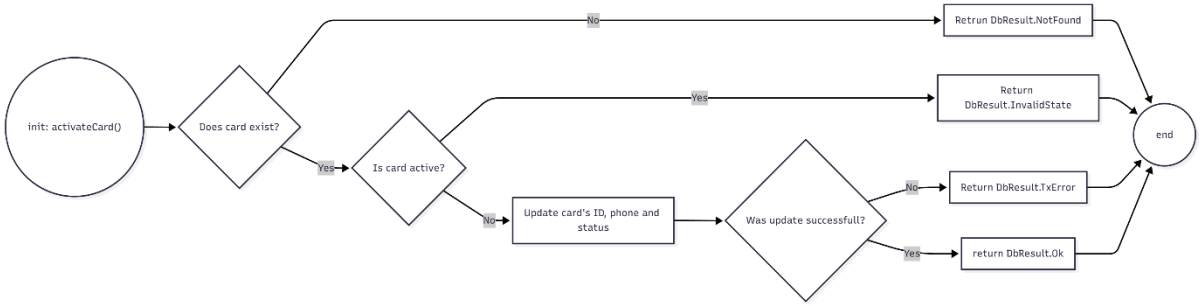

Figure 34 - activateCard() Function Flowchart

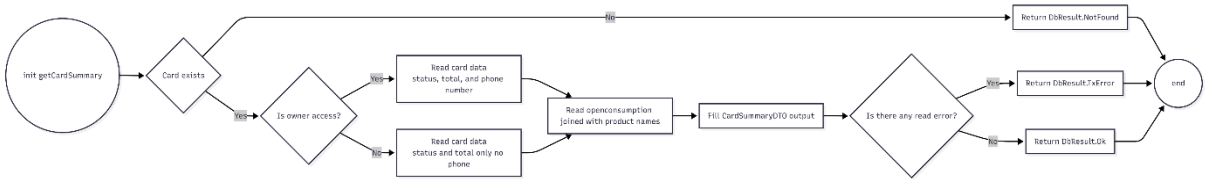

Figure 35 - getCardSummary() Function Flowchart

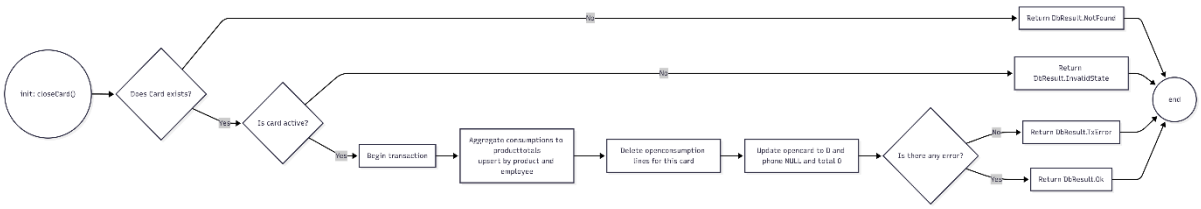

Figure 36 - closeCard() Function Flowchart

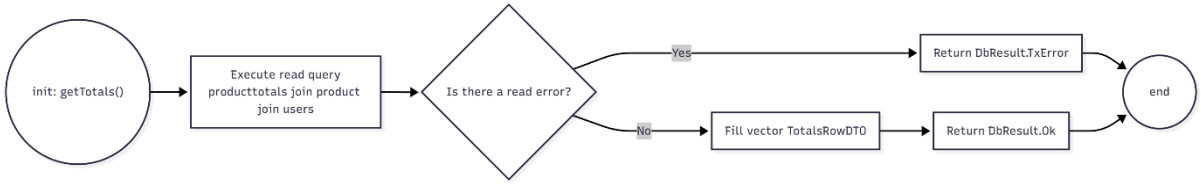

Figure 37 - getTotals() Function Flowchart

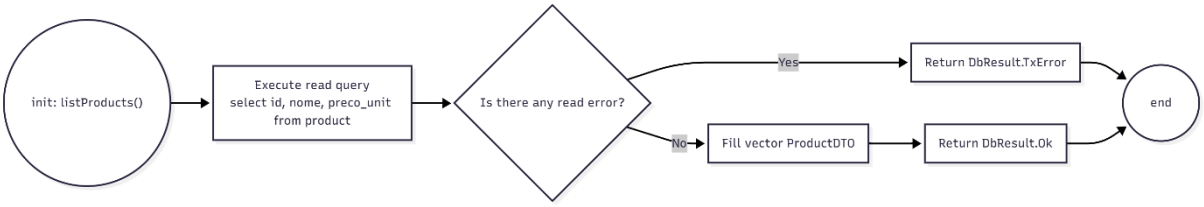

Figure 38 - listProducts() Function Flowchart

5.5.2. CardService flowcharts

On the following figures (Figure 39 to Figure 43), we describe how we will develop the CardService's functions, that bridge the gap between the classes ApiController and Database.

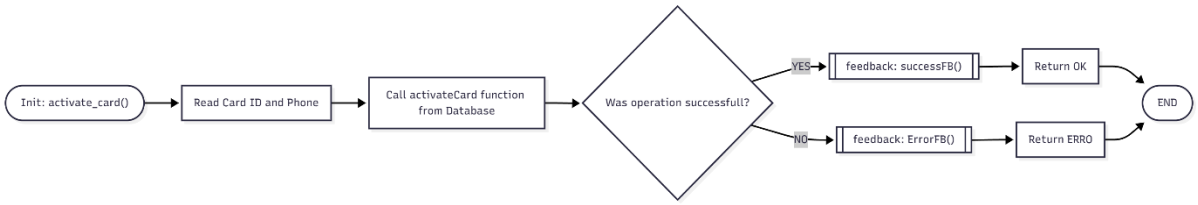

Figure 39 - activate_card() Function Flowchart

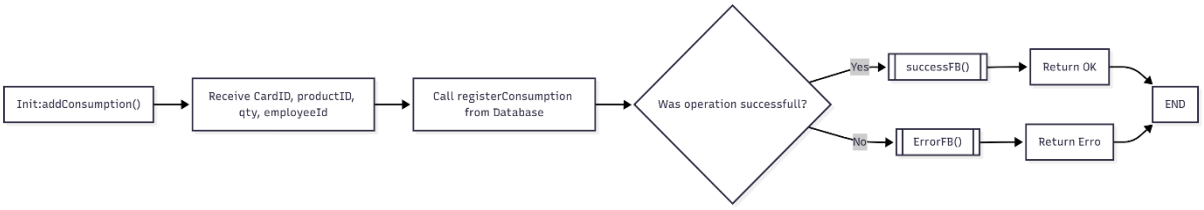

Figure 40 - addConsumption() Function Flowchart

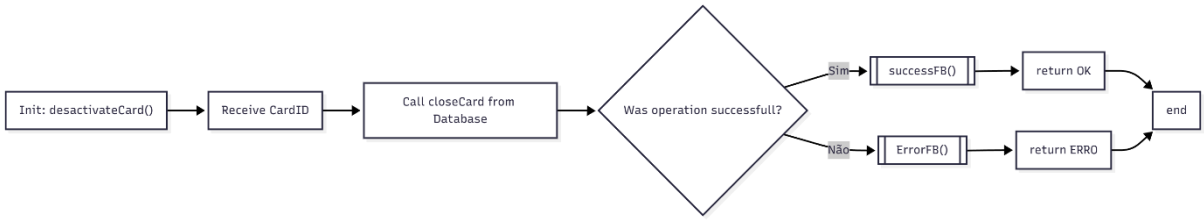

Figure 41 - deactivateCard() Function Flowchart

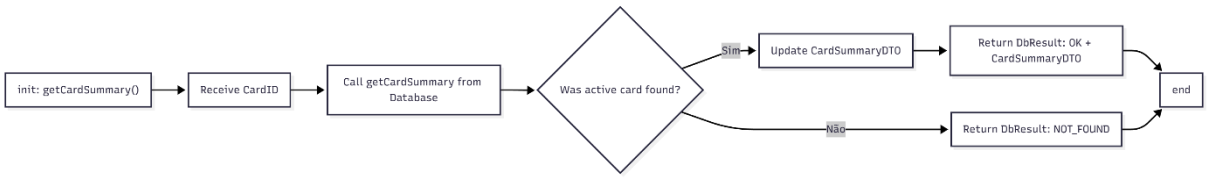

Figure 42 - getCardSummary() Function Flowchart

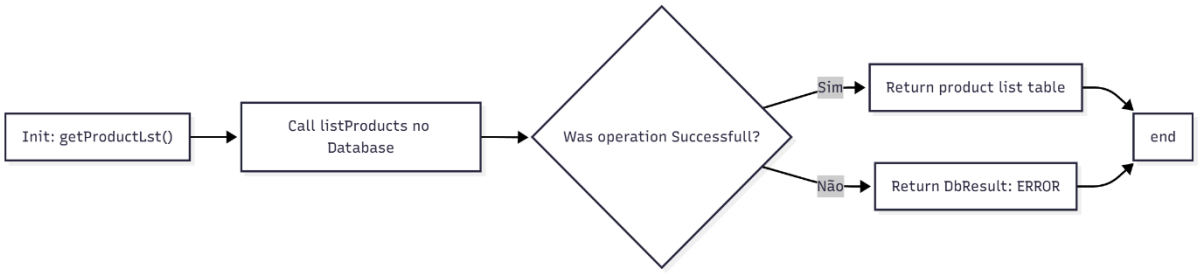

Figure 43 - getProductList() Function Flowchart

5.5.3. FeedbackController flowcharts

In following figures (Figure 44 to Figure 47), we describe how FeedbackController's functions, manage LEDs and buzzers according to system's feedback.

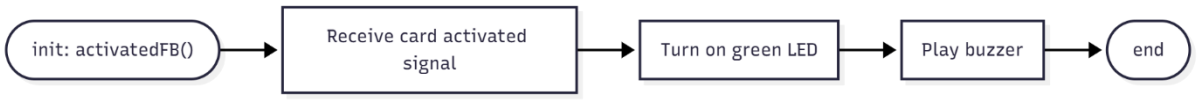

Figure 44 - activatedFB() Function Flowchart

Figure 45 - deactivateFB() Function Flowchart

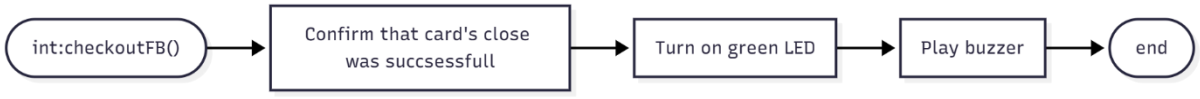

Figure 46 - checkoutFB() Function Flowchart

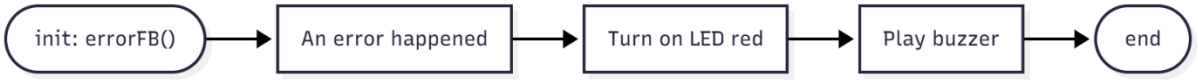

Figure 47 - errorFB() Function Flowchart

5.5.4. ApiController flowcharts

The following figures illustrate how the DbResult start() (Figure 48) and void stop() (Figure 49) functions of the ApiController class work, which are responsible for starting and ending system threads in a controlled manner.

The remaining functions of this class are associated with synchronization between threads and will be detailed in the following sections, which describe their joint behavior in the event flow and internal communication.

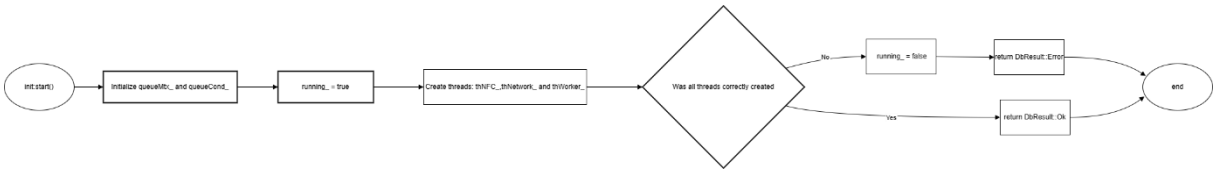

Figure 48 - start() Function Flowchart

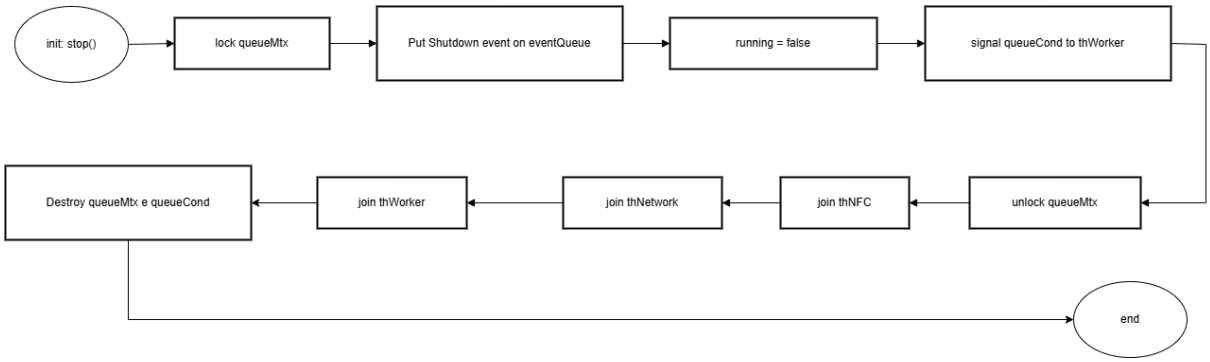

Figure 49 - stop() Function Flowchart

5.6. Threads

The NexiPass System is designed for a multithreaded system, using the POSIX pthreads library to ensure predictable response times.

5.6.1. Threads Overview

The system has three main threads created and managed by the class, each with responsibilities and priorities as shown in the following table:

Thread	Function	Prioridade
thNFC_	Captures NFC read events	High
thNetwork_	Processes requests from the mobile application	Medium
thWorker_	Manages the event queue and performs operations in the CardService and Database classes.	Low

Table 5 - Thread Functions and Priority Table

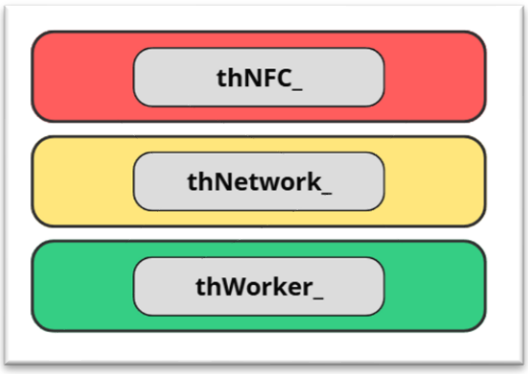

Figure 50 - Diagram of Thread Priority Hierarchy

5.6.2. Threads Behavior

Each thread executes a set of specific tasks and synchronizes itself through mutexes and conditional variables, thus avoiding race conditions when accessing the shared queue.

The functions of the threads are:

- **thNFC_**:
    - Monitors the NFC reader.
    - Creates ActivateCard and DeactivateCard events and places them in the queue.
    - Wakes up the thWorker thread using conditional variables.

- **thNetwork_**:
    - Receives HTTP requests from the APP.
    - Converts each request into an event and inserts that event into the eventQueue.
    - Ensures that multiple requests are processed in an orderly manner using a mutex.
- **thWorker_**:
    - Consumes the event queue in FIFO order
    - Calls the CardService functions corresponding to each event type, which in turn will read or save data in the database.
    - Communicates the results to the FeedbackController to activate the LED/Buzzer.

5.6.3. Thread Synchronization

As we can see in the following figure(Figure 51), which represents the flow of events and synchronization between threads in this system, the thNFC_ and thNetwork_ threads are event producers, while thWorker_ is the consumer thread, responsible for processing the business logic.

Communication between them is done through an event queue (eventQueue_) protected by a mutex (queueMtx_) and synchronized with a condition variable (queueCond_).

The producer threads block the mutex to insert new events into the queue and then signal the condition variable to wake up thWorker_, which is suspended while the queue is empty.

This mechanism ensures mutual exclusion and avoids busy waiting, ensuring deterministic and efficient execution. After waking up, thWorker_ processes the event by invoking CardService, which interacts with the Database and FeedbackController to update the system state and provide visual or audible feedback.

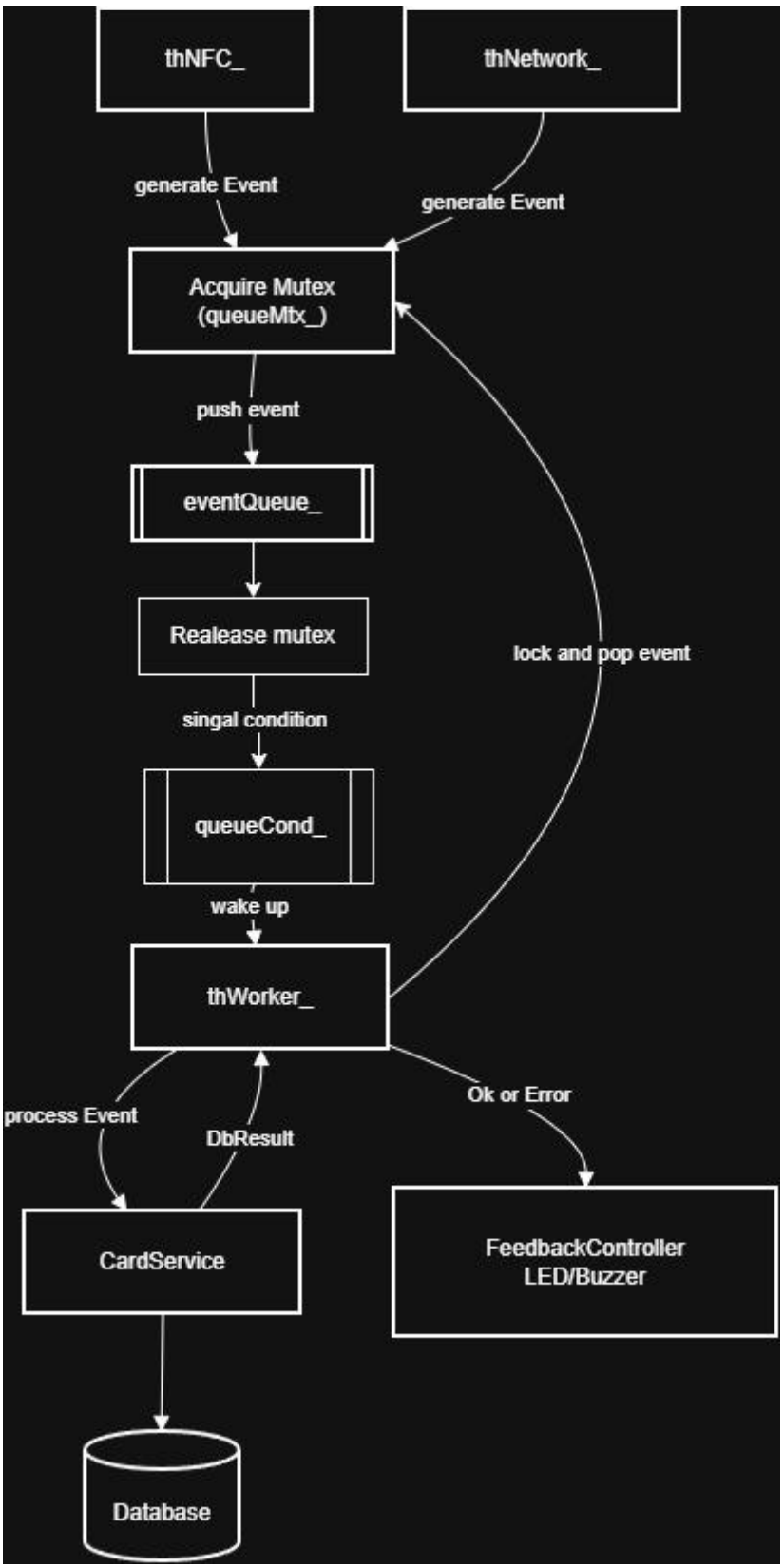

Figure 51 - Diagram of Thread Communication and Event Processing Flow

5.7. Application

To communicate with the board, we intend to develop an application with remote access. This application must be able to receive and send requests to a server, with the following behavior

The image illustrates the client-server architecture of the system, where clients (APP) communicate with the server (Board) through HTTP requests.

The server (Board) is responsible for managing operations on NFC cards and associated consumption, while clients only send requests and display results according to user actions.

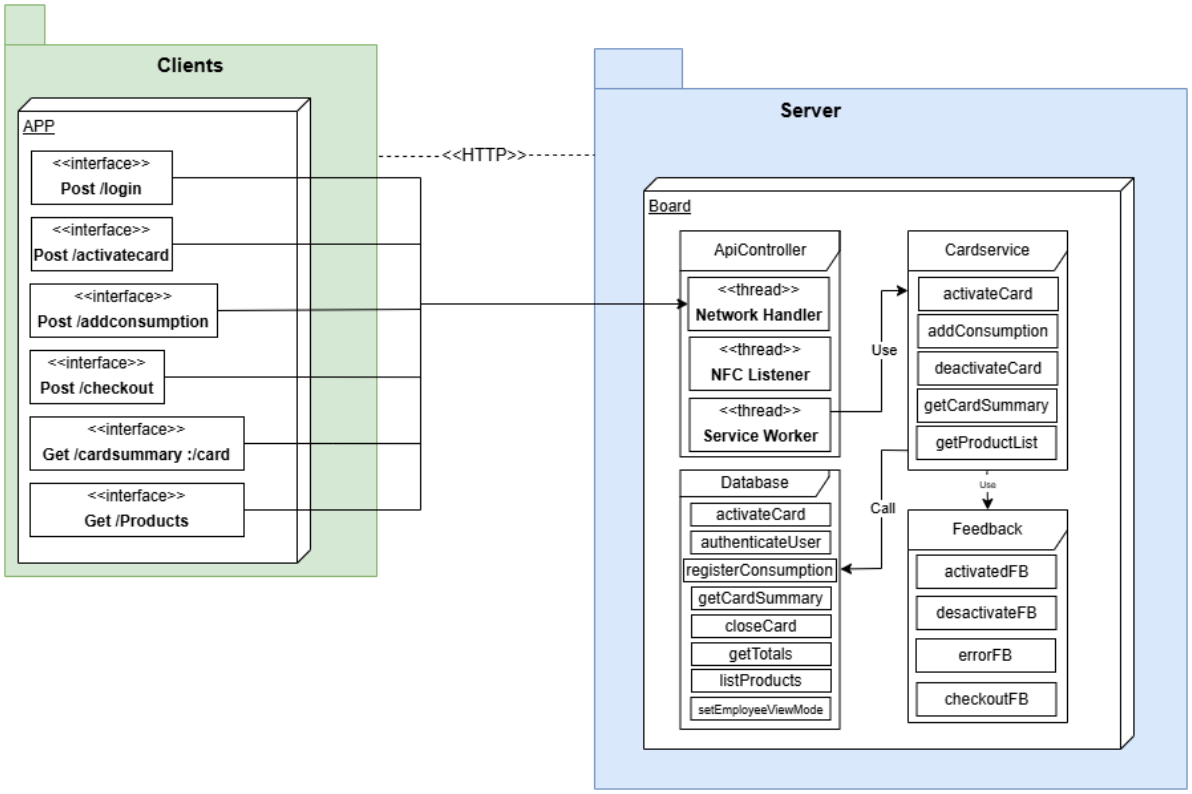

Figure 52 - UML Component Diagram – Client-Server Architecture

5.7.1. Clients (APP)

The green block represents the client application, a mobile application that provides the user with access to the main features of the system. To do this, the user has different HTTP interfaces available:

POST /login → authenticates the user.

POST /activatecard → activates an NFC card.

POST /addconsumption → records consumption associated with a card.

POST /checkout → ends the card consumption session.

GET /cardsummary/:card → obtains a summary of a card's consumption.

GET /products → returns the list of available products.

These endpoints are invoked by the user through the application's GUI interface and send requests to the server.

5.7.2. Server (Board)

The blue block represents the board that acts as a server, responsible for processing requests received via HTTP and performing internal operations that interact with the database, the NFC system, and the feedback logic.

The ApiController module acts as an input controller on the board, receiving client requests and using CardService methods to handle each request.

To do this, the module uses three distinct threads:

Network Handler – responsible for managing network communications (HTTP)

NFC Listener – listen to and processes events from the NFC reader

Service Worker – coordinates asynchronous tasks between different services

5.7.3. Gui Specification

The goal is to create a detailed structure for the design and functionalities of the graphical user interface for the mobile application. These interfaces must be simple and intuitive for the end user, while still complying with the system's logic.

5.7.3.1 Login

Initially, a login request screen is required. Each account must be previously created in the database by the system administrator, assigning it a username and password.

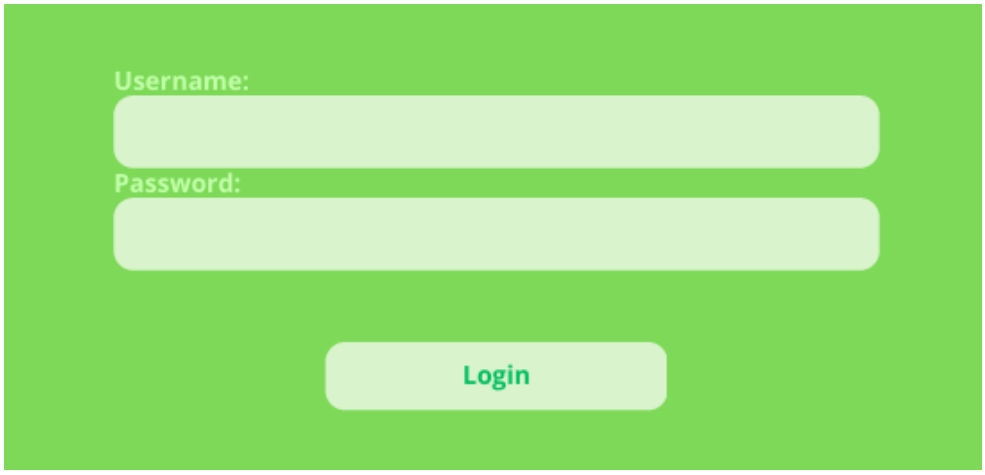

Figure 53 - User Login Screen Design

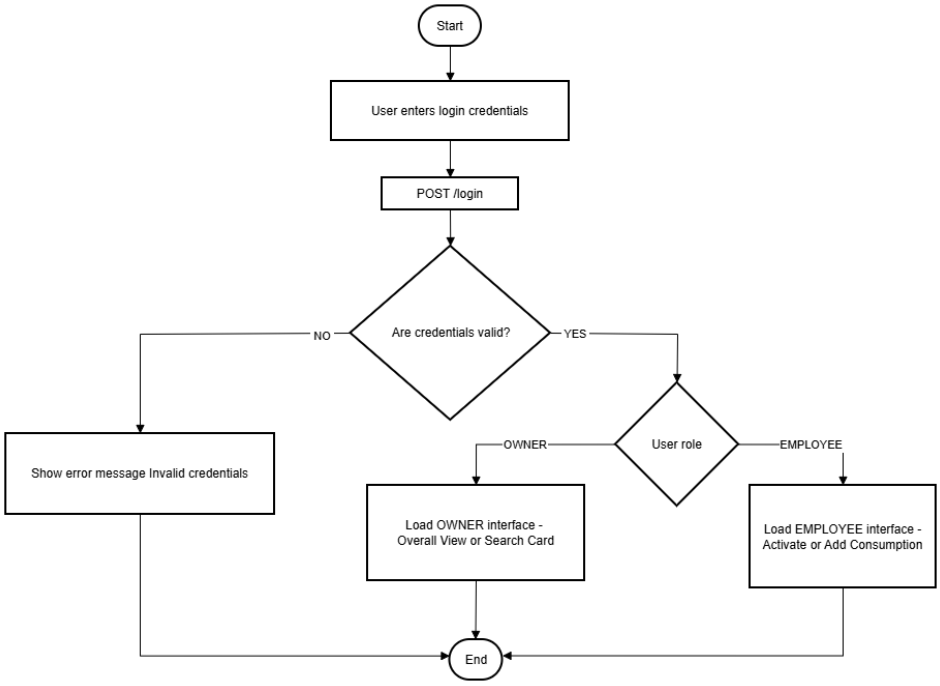

Figure 54 - Flowchart of the User Authentication Process

5.7.3.2 Card Activation

At this stage, the application must wait for the card to be read on a waiting screen and then move on to an activate and deactivate card interface, depending on the user role selected previously.

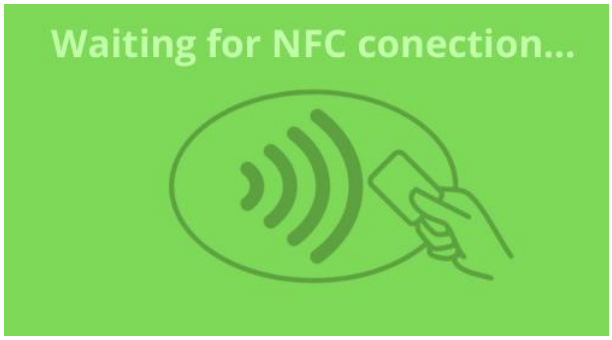

Figure 55 - User Interface – Waiting for NFC Connection

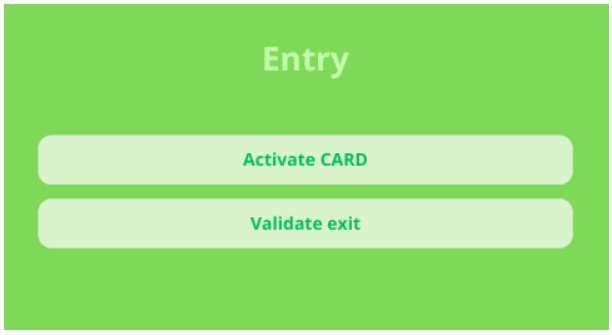

Figure 56 - User Interface – Waiting for NFC Connection

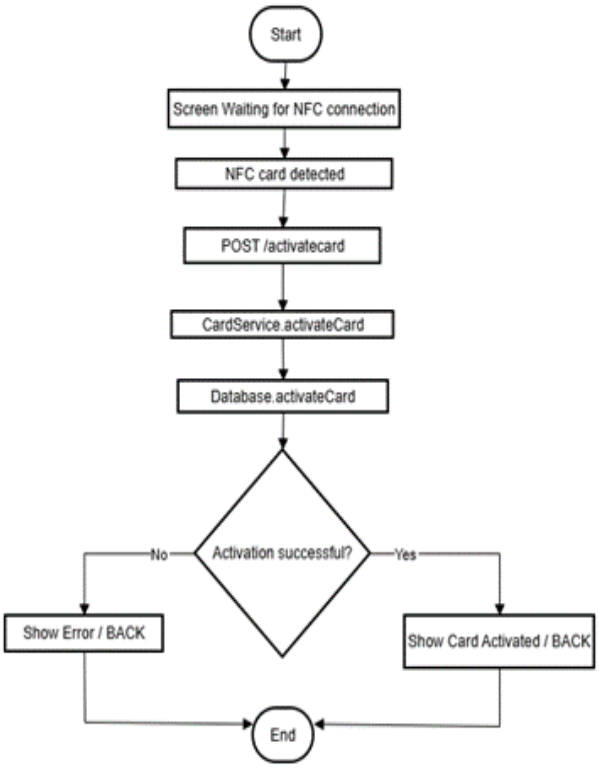

Figure 57 - Flowchart of the Card Activation Procedure

5.7.3.3 Add Consumption

In this section, if the selected user rule was POS, the system will wait for the NFC card to be read and then allow processes such as add consumption and view consumption.

Figure 58 - User Interface – Point of Sale Options

Figure 59 - User Interface – Add Consumption Functionality

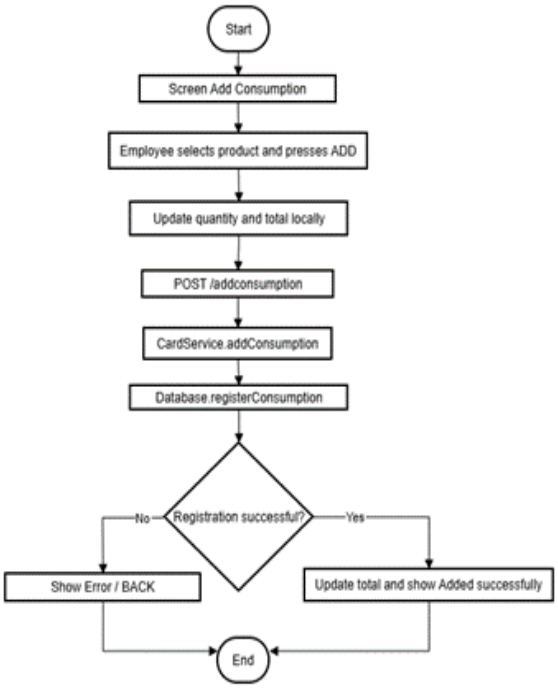

Figure 60 - Flowchart of the Add Consumption Procedure

5.7.3.4 Checkout

If the user rule is checkout, the user is able to perform functions such as deactivate card and view consumption, where it is possible to deactivate the card after payment and view data on the products consumed.

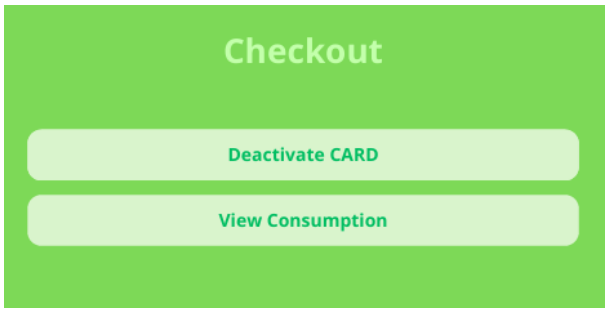

Figure 61 - User Interface – Checkout Options

Figure 62 - User Interface – Checkout Total and Card Deactivation

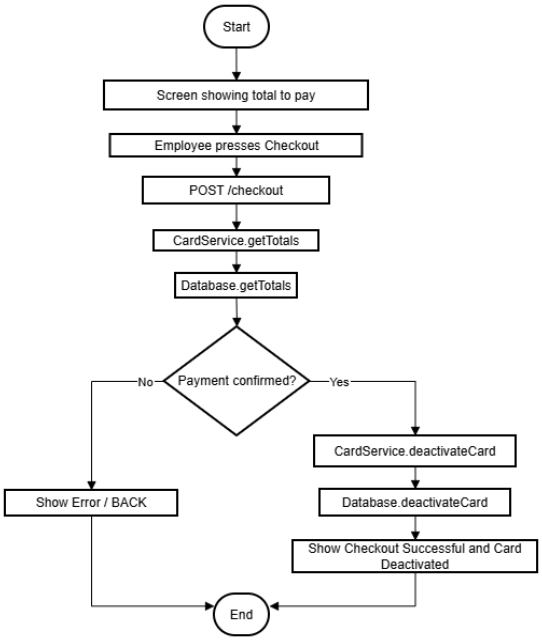

Figure 63 - Flowchart of the Checkout and Card Deactivation Process

5.7.3.5 View Consumption

A interface de view consumption, permite visualizar os artigos e valores associados ao cartão. Esta funcionalidade está disponivel no point of sale (POS) e no checkout.

Figure 64 - User Interface – Individual Consumption View

Figure 65 - User Interface – Global Consumption Overview

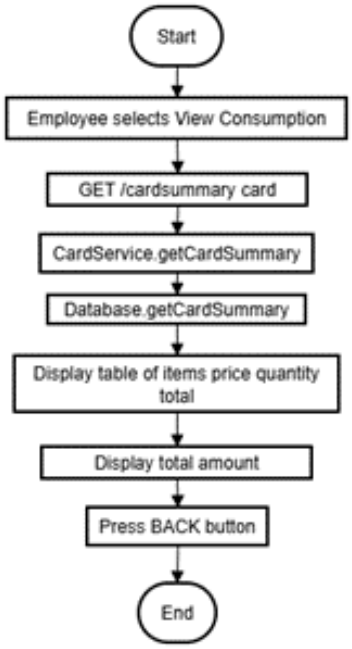

Figure 66 - Flowchart of the View Consumption Procedure

5.8. Test cases

Table 6 summarises the main tests planned for the post-implementation phase and their expected behaviour.

Any error that may occur will be managed by returning the appropriate error code defined in the DbResult(Figure *24*) enumeration, ensuring consistent and controlled error handling across the system.

Event	Expected Behavior
User login	There is user data on database→ log user →return DbResult::ok
Activate card	Update card status to Activated, turn on green LED and buzzer → return DbResult::ok
Deactivate card	Update card status to Deactivated, turn on red LED and buzzer → return DbResult::ok
Add Consumption	Add consumption for an activated card → return DbResult::ok
Get Summary	View an activated card Consumption → return DbResult::ok
Get Products	View products that exits on database → return DbResult::ok

Table 6 - test cases table

5.9. Software COTS

To develop our project, we use are going to use software tool described below.

5.9.1. Buildroot

Buildroot is used to create a custom Linux image for Raspberry Pi. It allows only the necessary packages, ensuring a lightweight operating system optimized, with faster boot times.

Figure 67 - Buildroot Logo

5.9.2. Visual Studio Code (VSCode)

VSCode is a lightweightand fast IDE used for editing, debugging, and building C++ code for this system.

Figure 68 - Visual Studio Code Logo

5.9.3. Flutter

Flutter is a framework used to develop applications for Android, iOS, web and desktop from a single codebase using Dart language. It was chosen to develop our APP interface due to its efficiency and versatility.

Figure 69 - Flutter Logo

5.9.4. Canva

Canva is a online graphic design platform that allows users to create visual content in an intuitive and accessible way. Its used to develop our interface prototype.

Figure 70 - Canva Logo

5.9.5. Draw.io

Draw.io is a platform used to design flowcharts, organizational charts, network diagrams, UML diagrams and more.

Figure 71 - Draw.io Logo

5.9.6. PgAdmin4

PgAdmin 4 is a free graphical tool for managing and developing PostgreSQL databases. It allows you to create, edit, and query database objects with an intuitive interface and advanced features such as an SQL editor, performance monitoring, and user management.

Figure 72 - pgAdmin Logo

5.9.7. pthreads

The use of the pthreads library in this system is due to the need to implement a deterministic and efficient multithreaded system capable of processing multiple events in parallel, i.e., ensuring that tasks are executed safely and predictably, guaranteeing consistent and responsive behavior on the part of the system.

Figure 73 - POSIX Threads (pthreads) Logo

6. Implementation

The implementation of NexiPass was guided by strict requirements for temporal determinism and resource efficiency. The final solution was developed in C++17 on a custom embedded Linux image, ensuring full control over memory management and process scheduling.

This chapter details the system engineering following a bottom-up approach, from the manipulation of physical records in the Kernel to the availability of data via REST API.

6.1. Embedded Linux System generated in Buildroot

To generate the Linux image, buildroot was used, which allowed full control over the system libraries. This tool allowed the complete system image (Boot, RootFS) to be generated, compiled specifically for the board (Raspberry Pi 4B). The basis of the system is not a general-purpose distribution, but rather minimalist firmware built specifically for the target hardware.

6.1.1. Toolchain Configuration and Cross-compilation

The Raspberry Pi 4B employs an ARM Cortex-A72 (Aarch64) architecture. Consequently, development took place on an x86_64 host via a cross-compilation approach.

The toolchain was generated by Buildroot with the following technical specifications:

- **Compiler**: aarch64-buildroot-linux-gnu-g++, allowing the use of modern features of C++17.
- **Library C**: glibc, selected to ensure full compatibility with libpq (PostgreSQL) libraries and robust support for POSIX Threads (pthreads).
- **Optimization**: In the application's Makefile, the -O2 flag was applied. This optimization is critical, removing dead code and optimizing interrupt vectors without compromising stability. The application compilation process is managed by a dedicated Makefile, configured to use these specifications.

```
CXX = /home/brunoa/buildroot/buildroot-2025.02.6/output/host/bin/aarch64-buildroot-linux-gnu-g++

CXXFLAGS = -std=c++17 -Wall -pthread -O2 -I.
LDFLAGS = -lpq -pthread

SRCS = $(wildcard *.cpp)
OBJS = $(SRCS:.cpp=.o)

TARGET = nexipass_app

all: $(TARGET)
$(TARGET): $(OBJS)
    $(CXX) $(OBJS) -o $(TARGET) $(LDFLAGS)

```

Figure 74 - Makefile crosscompile

6.1.2. libraries and packages

The operating system image was configured via **make menuconfig** in Buildroot to include only what was strictly necessary:

To prepare the image for the target architecture, the command make **raspberrypi4_64_defconfig** was used, ensuring the correct configuration for the final target.

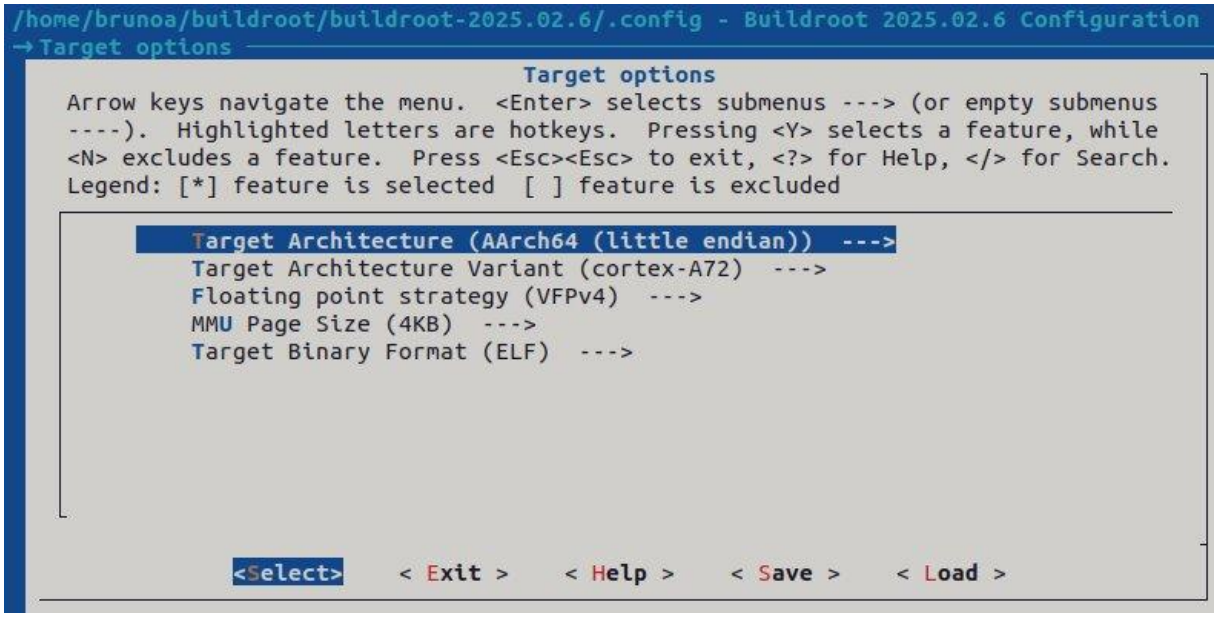

Figure 75 - Buildroot Target Access control in frontend: Conditional widget rendering based on currentUserRoleoptions (raspberrypi4_64_defconfig)

6.1.2.1 Bibliotecas de terceiros

It was necessary to activate some third-party libraries and packages to correctly respond to the system's needs. To do this, the following settings were activated:

- **Packages -> Hardware handling -> [*] spi-tools:** The spi-tools package allows to test and configure the SPI (Serial Peripheral Interface) bus directly from the command line, which is essential for debugging.

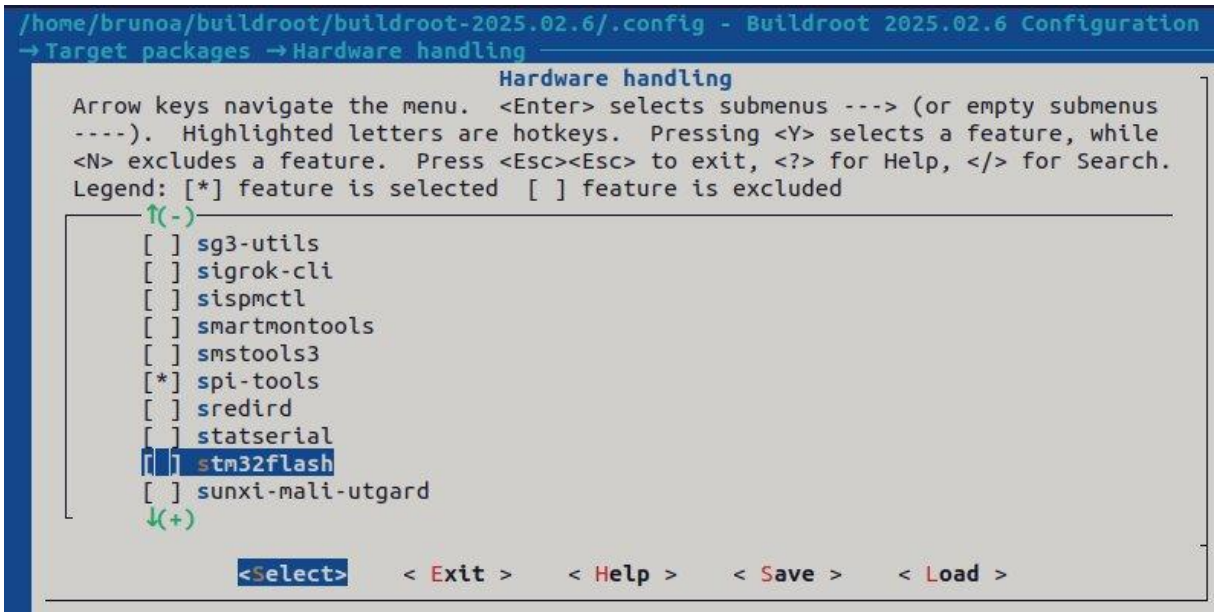

Figure 76 - Buildroot package selection: Enabling spi-tools for SPI bus debugging and testing in target system

- **Libraries -> Database -> [*] postgresql:** This configuration installs the native client library (libpq) on the target file system, providing the API necessary for the application to establish connections and execute SQL transactions on a remote server.

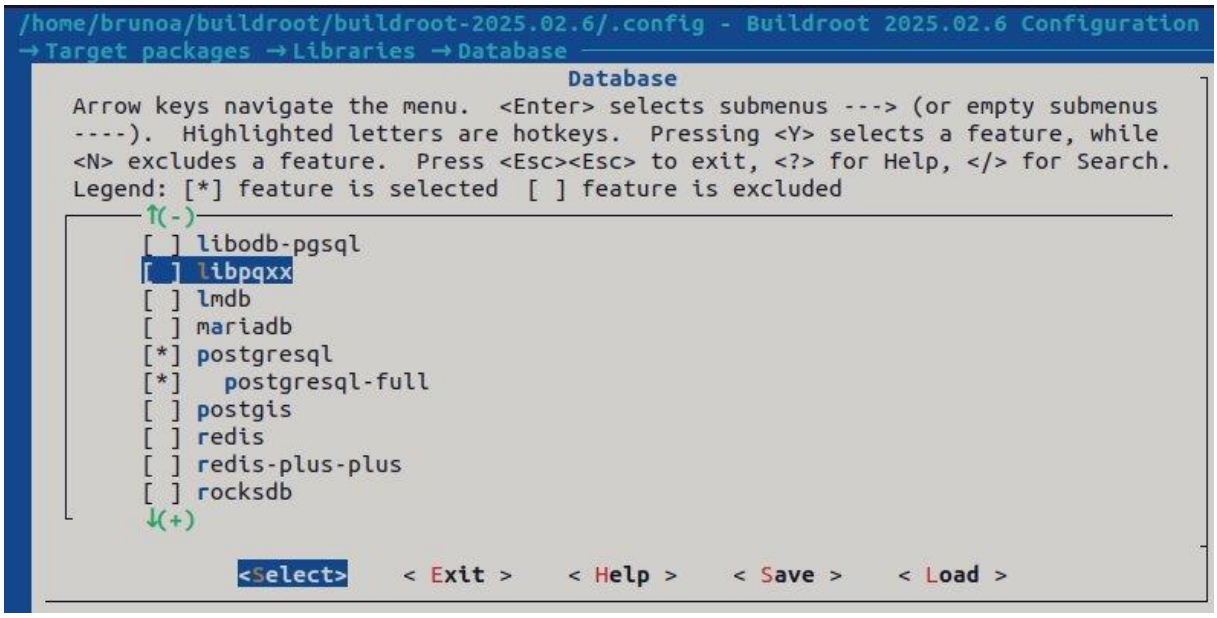

Figure 77 - Buildroot library configuration: PostgreSQL client library (libpq) integration for database connectivity

- **Packages -> Libraries -> Hardware handling -> [*] libgpiod :** This library provides a modern C++ API for interacting with the Linux kernel GPIO subsystem.

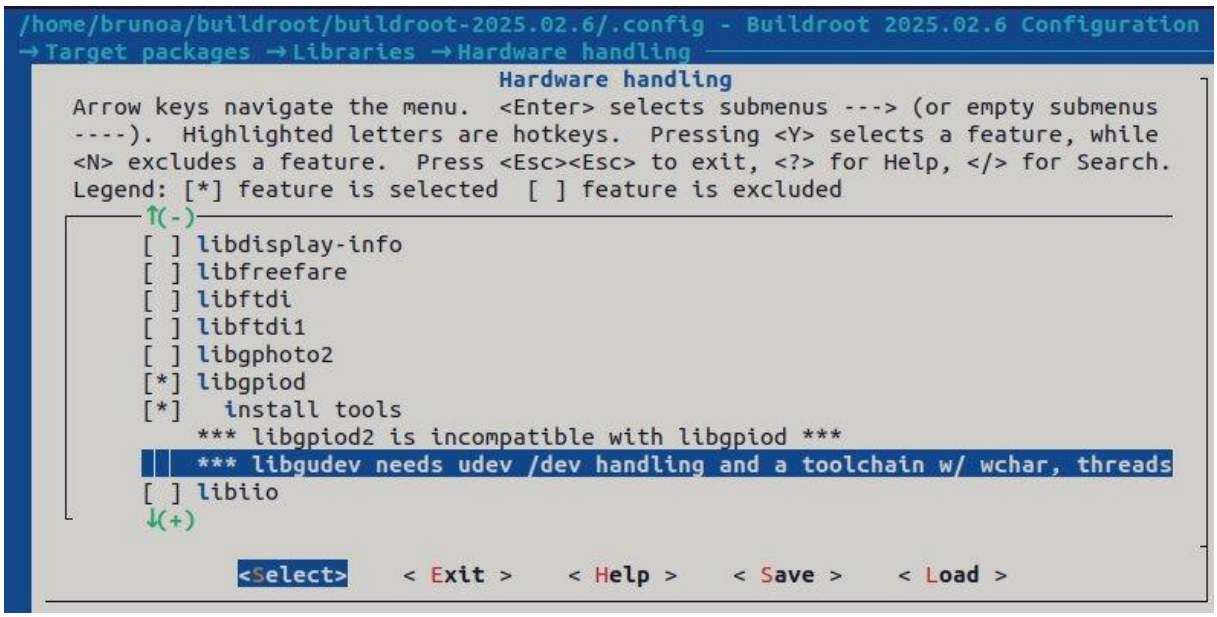

Figure 78 - Buildroot GPIO configuration: libgpiod library enabling modern kernel GPIO subsystem access

- **Libraries -> JSON/XML -> [*] rapidjson :** This is a C++ library for parsing and generating JSON, essential for exchanging structured messages between the ApiController and the mobile application.

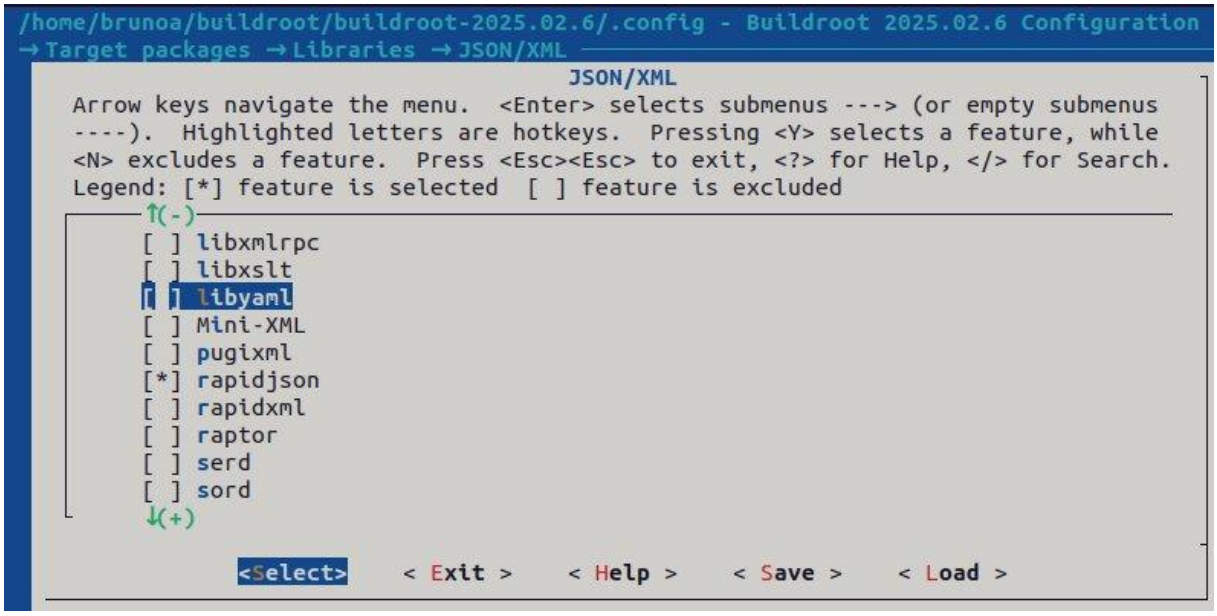

Figure 79 - Buildroot Json library

- **Validation of libraries and utilities installed in the final Buildroot image:** The test confirms the presence of the HTTP client (curl), GPIO controller detection (gpiodetect), database libraries (libpq), and network connectivity to the outside world (ping).

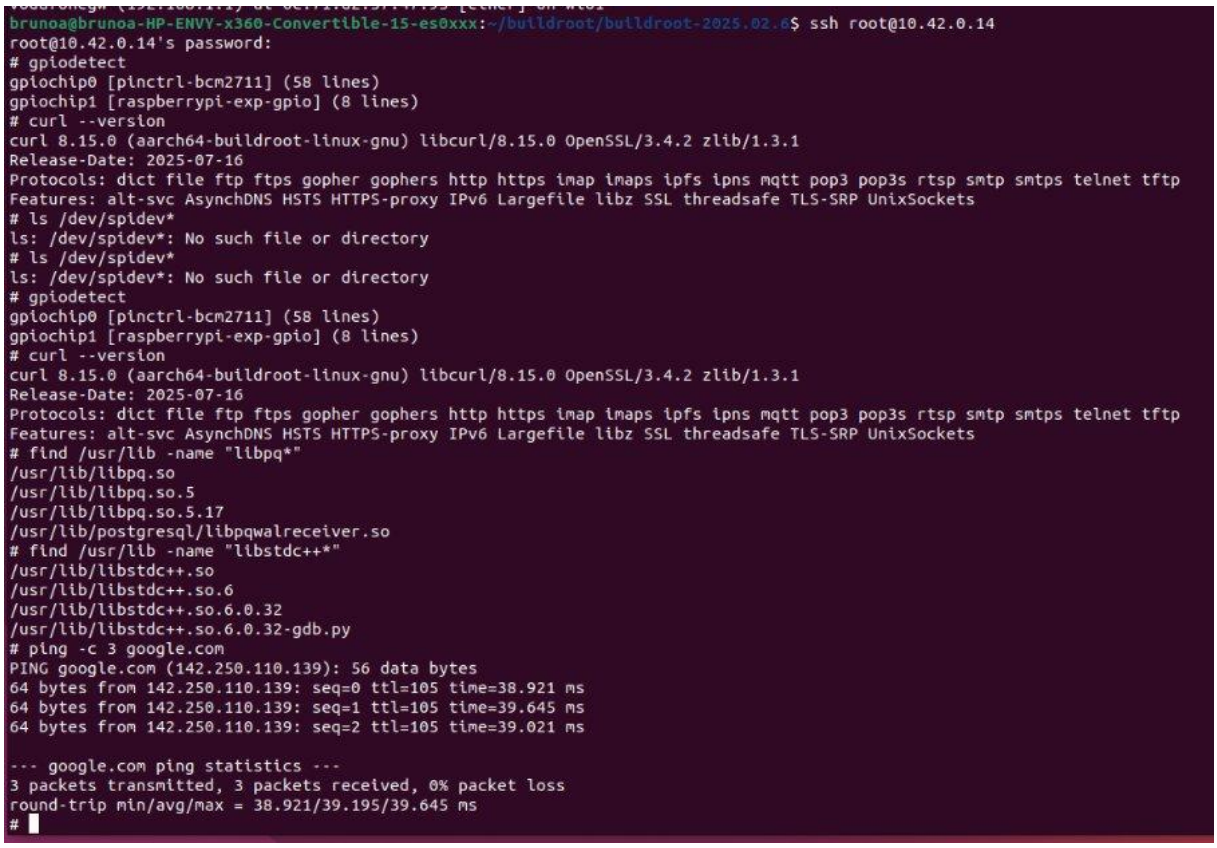

Figure 80 - Target system validation: Command-line output showing installed tools (curl, gpiodetect), libpq presence, and network connectivity test

6.1.3. Configuração do Boot e Device Tree

The correct initialization of peripherals is defined before the kernel is loaded, through the /boot/config.txt file, ensuring exclusive access to physical buses when loading the kernel.

The configurations applied were:

- **SPI interface (dtparam=spi=on):** Enables the SPI0 bus. This instructs the kernel to load the spidev driver and create the /dev/spidev0.0 node, used by the SimpleRFID class to communicate with the RFID reader.
- **PWM interface (dtoverlay=pwm,pin=13,func=2):** Buzzer control requires a precise square wave. Instead of generating the signal via software (which would consume excessive CPU cycles), the PWM hardware overlay was loaded onto GPIO pin 13. This exposes the /sys/class/pwm/pwmchip0 interface, allowing the FeedbackController to set the period and duty cycle of the sound without blocking the execution of the main application.

```
31 dtparam=spi=on
32 dtoverlay=pwm,pin=13,func=4
```

Figure 81 - config.txt vconfiguration for spi and pwm

6.2. Device Drivers

Developed for the visual feedback subsystem (Led RGB), which requires determinism and absolute control over the hardware, we opted to develop a proprietary Character Device Driver (ledrgb.ko).

This approach allows the complexity of the hardware to be isolated in Kernel Space, exposing only a simple and secure interface for the application in User Space.

6.2.1. Device driver registration and identification

The integration of the driver into the VFS (Virtual File System) subsystem begins with the allocation of a unique identifier. The code uses the alloc_chrdev_region function to dynamically obtain a Major Number and a Minor Number.

- **Major number:** Identifies the driver associated with the device (driver ledrgb)
- **Minor number:** Identifies the specific instance of the device (useful for supporting multiple LED controllers)

After allocation, the driver registers the cdev (Character Device) structure, making it visible to the kernel. For the device to be accessible in User Space without manually creating nodes (via mknod), automatic creation of the class and device was implemented:

```
107  static int __init
108      led_init (void){
109
110      int ret;
111      struct device *dev_ret;
112
113      //dynamic allocation of Major/Minor number
114      if ((ret = alloc_chrdev_region(&DEV_major_minor, 0, NUM_DEVICES, DEVICE_NAME)) != 0){
115          printk(KERN_DEBUG "Can't register device\n"); return ret;
116      }
117
118      //create device class
119      if (IS_ERR(led_class = class_create(CLASS_NAME))){
120          unregister_chrdev_region (DEV_major_minor, NUM_DEVICES);
121          return (PTR_ERR(led_class));
122      }
123
124
125      //create device node
126      if (IS_ERR(dev_ret = device_create(led_class, NULL, DEV_major_minor, NULL, DEVICE_NAME))) {
127          class_destroy(led_class);
128          unregister_chrdev_region (DEV_major_minor, NUM_DEVICES);
129          return PTR_ERR(dev_ret);
130      }
131
132      //initialize the char device and link it with fops
133      cdev_init(&c_dev, &ledDevice_fops);
134      c_dev.owner = THIS_MODULE;
135
136
137      //add the device to the system
138      if ((ret = cdev_add(&c_dev, DEV_major_minor, NUM_DEVICES)) < 0) {
139          printk(KERN_NOTICE "Error %d adding device", ret);
140          device_destroy(led_class, DEV_major_minor);
141          class_destroy(led_class);
142          unregister_chrdev_region(DEV_major_minor, NUM_DEVICES);
143          return ret;
144      }
145
146      /* ioremap: enforce mem protection: given addr is physical, it maps onto
147          the virtual memory addr. */
148      s_GPIO = (GPIORegister *)ioremap(GPIO_BASE_ADDR, sizeof(GPIORegister));
149      pr_alert("Map to virtual address: %p\n", s_GPIO);
150
151      // Configure RGB pins as outputs
152      SetGPIOFunction(s_GPIO, PIN_RED, OUTPUT);
153      SetGPIOFunction(s_GPIO, PIN_GREEN, OUTPUT);
154      SetGPIOFunction(s_GPIO, PIN_BLUE, OUTPUT);
155
156      SetGPIOState(s_GPIO, PIN_RED, 0);
157      SetGPIOState(s_GPIO, PIN_GREEN, 0);
158      SetGPIOState(s_GPIO, PIN_BLUE, 0);
159      return 0;
160  }
```

Figure 82 - Kernel driver registration: Character device allocation using alloc_chrdev_region, class_create, and device_create for automatic /dev/ledrgb node generation

6.2.2. File_operations structure and Inodes

The interaction between the application and the driver is defined by the file_operations (fops) structure.

This table maps standard Linux system calls (open, write, release) to internal driver functions.

When the application executes open("/dev/ledrgb"), the kernel calls the led_device_open function, passing it two critical structures:

1. **Struct inode:** Represents the physical file in memory, contains the i_cdev field that points to the driver
2. **Struct file:** Represents the open instance of the file. The driver can use the "private_data" field to store state between calls.

```
100  static struct file_operations ledDevice_fops  = {
101          .owner = THIS_MODULE,
102      .write = led_device_write,
103      .read = led_device_read,
104      .release = led_device_close,
105      .open = led_device_open,
106  };
```

Figure 83 - VFS integration: file_operations structure mapping system calls (open, write, release) to ledrgb driver functions

```
32  int led_device_open (struct inode* p_indode, struct file *p_file){
33      pr_alert("%s: called\n",__FUNCTION__);
34      p_file->private_data = (GPIORegister *) s_GPIO;
35      return 0;
36  }
37
```

Figure 84 - Driver open handler: Initialization routine storing GPIO register pointer in file->private_data for subsequent write operations

6.2.3. Driver Architecture and Memory Management (MMIO)

The module was developed for the BCM2711 architecture (Raspberry Pi 4). The central challenge in this architecture is that access to peripherals is done via Memory Mapped I/O, but the Kernel operates in a protected virtual address space.

The architecture of the visual feedback subsystem, illustrated in Figure 85, demonstrates the strict separation between User Space and Kernel Space. The control flow follows three critical stages:

Abstract Interface: The NexiPass application does not access memory addresses directly. Interaction is performed through the Linux Virtual File System (VFS), writing to the /dev/ledrgb device file created by the driver.

Context Switch: When executing the write () syscall, a context switch to kernel mode occurs. The driver (ledrgb.ko) uses the copy_from_user function to transfer data securely, preventing memory access violations.

Physical Access (MMIO): The driver bypasses virtual memory using the ioremap function. This creates a direct bridge to the physical base address 0xFE200000, allowing write operations to virtual registers (s_GPIO->GPSET) to be converted into electrical signals on the physical pins of the Raspberry Pi.

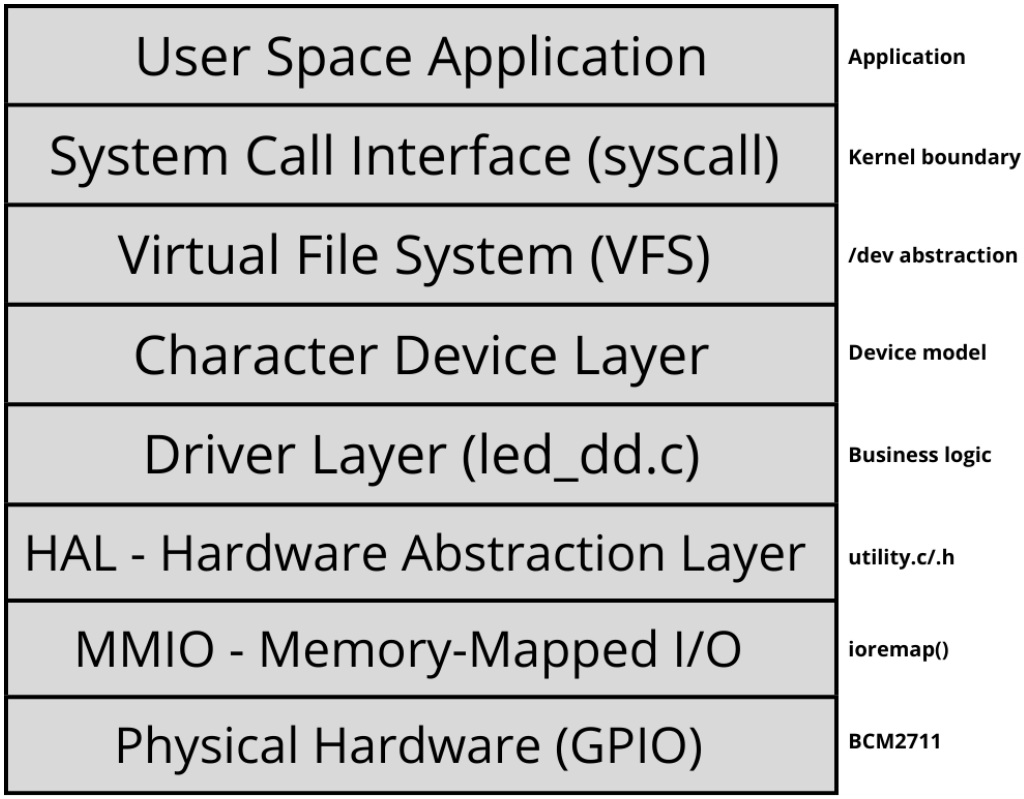

Figure 85 - Memory-mapped I/O architecture: Three-layer diagram showing User Space (application), Kernel Space (ledrgb.ko driver with ioremap), and Hardware (BCM2711 GPIO registers at physical address 0xFE200000)

```
// led_dd.c - Mapeamento de Memória Física para Virtual
#define GPIO_BASE_ADDR 0xFE200000 // Base de Periféricos do BCM2711 (RPi 4)

// Criação de ponteiro virtual para a região física
s_GPIO = (GPIORegister *)ioremap(GPIO_BASE_ADDR, sizeof(GPIORegister));
```

Figure 86 - physical base address 0xFE200000 and ioremap function

The GPIORegister structure (defined in utility.h) strictly maps the memory layout specified in the Broadcom datasheet, aligning the 32-bit registers:

GPFSEL (Function Select): To configure pins as Input or Output.

GSET e GPCLR: To set high or low logic levels.

```
23  typedef struct{
24      uint32_t GPFSEL[NUM_SELECT_REG];
25      uint32_t Reserved0;
26      uint32_t GPSET[NUM_SET_CLR_REG];
27      uint32_t Reserved1;
28      uint32_t GPCLR[NUM_SET_CLR_REG];
29  } GPIORegister;
```

Figure 87 - GPIO register mapping: C struct mirroring BCM2711 hardware layout (GPFSEL, GPSET, GPCLR registers) for type-safe memory access

6.2.4. Hardware abstraction layer (utility.c)

To manipulate the hardware, direct bitwise operations were required in the mapped memory to implement the control logic.

In the **utility.c** file, the **SetGPIOState** function is used to write directly to the **GPSET** and **GPCLR** registers using bitwise operations. The operation is atomic at the hardware level, avoiding the need for Read-Modify-Write cycles.

```
22  void SetGPIOState (GPIORegister* s_GPIOregister, const int GPIO_PinNum, const bool outputValue){
23      if (!s_GPIOregister || CHECK_PIN(GPIO_PinNum) != 0)
24          return;
25
26      //Calcula qual o banco de registos (0-31 ou 32-57)
27      uint8_t x = CHOOSE_X_GPxxxX(GPIO_PinNum);
28      if (outputValue)
29          s_GPIOregister->GPSET[x] = 1 << (GPIO_PinNum % 32);
30      else
31          s_GPIOregister->GPCLR[x] = 1 << (GPIO_PinNum % 32);
32  }
```

Figure 88 - Atomic GPIO manipulation: Bitwise operations on GPSET/GPCLR registers

6.2.5. Input validation and business logic

The led_device_write function implements the driver control interface and is invoked whenever the application writes to the /dev/ledrgb device file. The flow begins with payload validation and secure transfer of data from User Space to a local buffer in the Kernel using

copy_from_user. After verifying that the sequence contains only valid binary characters (‘0’ or ‘1’), the function performs positional parsing of the three bytes, mapping them directly to the physical pins (R, G, B) and changing their electrical state using the auxiliary function SetGPIOState.

```
49  ssize_t led_device_write(struct file *pfile, const char __user *buf, size_t len, loff_t *offset){
50
51      pr_alert("%s: called (%zu)\n",__FUNCTION__,len);
52
53      if (unlikely(pfile->private_data == NULL))
54          return -EFAULT;
55
56      if (len < 3) {
57          pr_warn("ledrgb: insufficient data, need 3 bytes (RGB)\n");
58          return -EINVAL;
59      }
60
61      char kbuf[3];
62      if (copy_from_user(kbuf, buf, 3))
63          return -EFAULT;
64
65      if ((kbuf[0] != '0' && kbuf[0] != '1') ||
66          (kbuf[1] != '0' && kbuf[1] != '1') ||
67          (kbuf[2] != '0' && kbuf[2] != '1')) {
68          pr_warn("ledrgb: invalid input, expected '0' or '1' for each RGB component\n");
69          return len;
70      }
71
72      GPIORegister* pdev = (GPIORegister*) pfile->private_data;
73
74      // Red
75      if (kbuf[0] == '0')
76          SetGPIOState(pdev, PIN_RED, 0);
77      else
78          SetGPIOState(pdev, PIN_RED, 1);
79
80      // Green
81      if (kbuf[1] == '0')
82          SetGPIOState(pdev, PIN_GREEN, 0);
83      else
84          SetGPIOState(pdev, PIN_GREEN, 1);
85
86      // Blue
87      if (kbuf[2] == '0')
88          SetGPIOState(pdev, PIN_BLUE, 0);
89      else
90          SetGPIOState(pdev, PIN_BLUE, 1);
91
92      pr_info("ledrgb: Set RGB to %c%c%c\n", kbuf[0], kbuf[1], kbuf[2]);
93
94      return len;
95  }
```

Figure 89 - Driver write handler: Input validation, copy_from_user security, and RGB protocol parsing

6.2.6. User-Space Interface (/dev/ledrgb)

The driver registers itself in the system as a character device, dynamically creating the /dev/ledrgb node through integration with *udev*.

Communication follows the Unix paradigm “Everything is a File”, thus ensuring security, since the led_device_write function uses copy_from_user to transfer data from the application to the kernel securely, preventing invalid memory accesses.

A simple 3-byte protocol was also defined to control the colors of the RGB LED, respectively.

This abstraction simplifies implementation in the final application, which only needs to execute a standard **write ()**, completely abstracting itself from memory addresses or processor registers.

```
brunoa@brunoa-HP-ENVY-x360-Convertible-15-es0xxx:~$ ssh root@10.42.0.13
root@10.42.0.13's password:
# ls -l /dev/ledrgb
crw-------    1 root     root      240,   0 Jan  1 00:00 /dev/ledrgb
# grep 240 /proc/devices
240 ledrgb
```

Figure 90 - Device node verification: Command output showing /dev/ledrgb character device with major numbers assigned by kernel

6.3. Hardware abstraction Layer (HAL – user space)

Different from the implementation of the LED driver that operates at the kernel level, the remaining peripherals (RFID and buzzer) are managed in user space. The decision not to use kernel drivers for all hardware is motivated by the ease of development and debugging in user space, but on the other hand requires rigorous implementation to avoid typical high-level library latencies.

6.3.1. Driver RFID (SimpleRFID)

To communicate with the RFID sensor module, the SimpleRFID class was developed, which acts as a Linux driver in user space, serving as a bridge between the hardware and the logical system.

- **SPI protocol via ioctl:** Communication uses the native Linux interface /dev/spidev0.0. Data transmission is performed through transfer blocks configured in the spi_ioc_transfer structure and sent to the kernel through the ioctl system call. This allows communication with minimal CPU overhead.

- **Hardware state machine:** This class interacts directly with the /dev/spidev0.0 device via the ioctl syscall.

Communication occurs by sending structured packets (spi_ioc_transfer) directly to the RFID module registers.

- **CommandReg(0x01):** Used to send commands such as PICC_REQIDL (request to search for cards).
- **FIFODataReg(0x09):** Used to read the UID of the detected card.

```
uint8_t spiRead(uint8_t reg) {
    uint8_t tx[2];
    uint8_t rx[2] = {0};
    tx[0] = ((reg << 1) & 0x7E) | 0x80;
    tx[1] = 0x00;
    struct spi_ioc_transfer tr = {
        .tx_buf = (unsigned long)tx,
        .rx_buf = (unsigned long)rx,
        .len = 2,
        .speed_hz = speed,
        .bits_per_word = bits,
    };
    ioctl(spi_fd, SPI_IOC_MESSAGE(1), &tr);
    return rx[1];
}
```

Figure 91 - SPI communication implementation: ioctl-based data exchange using spi_ioc_transfer structure for register read/write operations

6.3.1.1 RFID module datasheet registers

The robustness of the SimpleRFID driver is based on direct mapping of the chip's control registers, as defined in the manufacturer's datasheet. SPI communication has been structured to handle critical memory addresses without abstraction overhead.

The main commands implemented for the machine were:

Register(Hex)	Name	Comando Enviado	Function
0x01	CommandReg	0x0E (softReset)	Reset the chip before starting the read cycle.
0x01	CommandReg	0x26 (PICC_REQIDL)	Sends the “Request Idle” request to wake up cards in the field.

0x09	FIFODataReg	Leitura de dados	Input/output buffer where the card UID is stored.
0x01	CommandReg	0x93 (PICC_ANTICOLL)	Start the anti-collision algorithm to isolate a single UID.

Table 7 - RFID register map: MFRC522 command/control registers

6.3.1.2 Reading and Validation Logic

The code performs the following technical steps:

1. **Polling (Request):** Sends the PICC_REQIDL command continuously.
2. **Anti-Collision:** When it detects an interruption, it executes the anti-collision algorithm to isolate the 4 byte UID.
3. **Synchronization**: Once the card's CRC has been validated, the UID is inserted into the workQueue protected by a mutex, notifying the Worker Thread to process the access.

```
199     bool isCardPresent() {
200         if (spi_fd < 0) return false;
201         uint8_t bufferATQA[2];
202         uint8_t bufferSize = sizeof(bufferATQA);
203         spiWrite(BitFramingReg, 0x07);
204         uint8_t cmd = PICC_REQIDL;
205         return (communicateCard(PCD_TRANSCEIVE, &cmd, 1, bufferATQA, &bufferSize) == 0);
206     }
207 
208     bool readCardUID(std::string &uid_hex) {
209         if (spi_fd < 0) return false;
210         uint8_t serNum[5];
211         uint8_t serNumLen = 5;
212         spiWrite(BitFramingReg, 0x00);
213 
214         uint8_t cmd[2] = {PICC_ANTICOLL, 0x20};
215 
216         if (communicateCard(PCD_TRANSCEIVE, cmd, 2, serNum, &serNumLen) == 0) {
217             if (serNumLen == 5) {
218                 uint8_t bcc = 0;
219                 for (int i = 0; i < 4; i++) bcc ^= serNum[i];
220 
221                 if (bcc == serNum[4]) {
222                     char hex[9];
223                     sprintf(hex, "%02X%02X%02X%02X",
224                             serNum[0], serNum[1], serNum[2], serNum[3]);
225                     uid_hex = std::string(hex);
226                     return true;
227                 }
228             }
229         }
230         return false;
231     }
232 };
```

Figure 92 - Anti-collision algorithm with PICC_REQIDL request and BCC (Block Check Character) validation

6.3.2. Audiovisual feedback (FeedbackController)

A critical error in real-time systems is the use of blocking functions such as sleep() to generate temporal patterns. If the main thread goes to sleep, the system becomes unable to detect new cards or process requests from the network during that interval.

The FeedbackController class solves this problem by implementing an asynchronous architecture based on advanced Linux kernel primitives:

- **Hardware Control:** The buzzer is controlled via the PWM (Pulse Width Modulation) subsystem of the BCM2711 hardware, exposed in /sys/class/pwm/pwmchip0.

  To ensure maximum acoustic efficiency of the piezoelectric transducer, the signal was configured with the following physical parameters:

  - **Period (T):** 250.000 ns
  - **Frequency (f):** 4kHz
  - **Duty Cycle:** 50%

- **Kernel timers(timerfd):** To eliminate the need for active or blocking waiting, the controller uses the timerfd_create system call. This API allows time counting to be delegated to the kernel, which notifies the application via a file descriptor.

- **Event Multiplexing (poll):** The feedback thread operates in an event multiplexing cycle using poll(). It remains suspended (Sleep state) and wakes up only in two situations:

  - **The timer has expired (timerFd_):** Indicates that the beep/silence time has ended and the hardware state must be changed.
  - **New Event (queueCv_):** The message queue received a priority request, immediately interrupting any running pattern.

6.4. Data Structures

To ensure the integrity of data circulating between the Hardware, Database, and API, the system defines strict structured types (DTOs) and error control enumerations, defined in Database.h.

6.4.1. Data Transfer Objects (DTOs)

Instead of unstructured arrays, structs are used to encapsulate logical entities. This facilitates JSON serialization and prevents type errors.

```
11  struct UserDTO {
12      uint32_t user_id;
13      string username;
14      string role;
15  };
16
17  struct ProductDTO {
18      int product_id = 0;
19      string name;
20      double price_unit = 0.0;
21  };
22
```

Figure 93 - Data Transfer Objects: C++ struct definitions mirroring PostgreSQL table schemas

6.4.2. Error Management (DbResult)

A strongly typed Enum Class (DbResult) was implemented. This allows the service layer to distinguish between a connection error (retry) and a business rule violation (abort).

- **DbResult::Ok**: Successful operation.
- **DbResult::ConnectionError**: Failure in the TCP socket of libpq.
- **DbResult::ConstraintFailed:** Integrity violation (e.g., negative balance).

```
enum class DbResult {
    Ok = 0,
    NotFound,
    InvalidState,
    ConstraintFailed,
    ConnectionError,
    TxError,
    UnknownError
};
```

Figure 94 - DbResult enum class

6.4.3. Implementation of the Data Layer and Persistence

The NexiPass data architecture was designed to ensure transactional integrity and information security in a multi-user environment. The system uses PostgreSQL as its database engine, interacting with the embedded software through a custom abstraction layer in C++.

6.4.3.1 Modelação de Dados Relacional

The database schema is not limited to passive storage; it imposes business rules at the structural level to ensure data quality.

To avoid state inconsistencies, custom data types (ENUM) were defined:

- **card_status ('A', 'D'):** Restricts the card status to Active or Inactive, preventing invalid states.
- **user_role ('OWNER', 'EMPLOYEE'):** Strictly defines the access hierarchy.

```
-- 1) TIPOS ENUM
CREATE TYPE user_role AS ENUM ('OWNER','EMPLOYEE');
CREATE TYPE card_status AS ENUM ('A','D'); -- A=Ativo, D=Desativado
```

Figure 95 - Type enumeration in Database

Core Tables and Optimization

The openconsumption table, responsible for recording transactions in real time, uses an advanced feature of PostgreSQL: Generated Columns. The line_total column is calculated automatically by the database engine itself:

```sql
CREATE TABLE openconsumption (
    id_consumo      SERIAL PRIMARY KEY,
    cartao_id       VARCHAR(16) NOT NULL REFERENCES opencard(card_id) ON DELETE CASCADE,
    product_id      INT NOT NULL REFERENCES product(product_id) ON UPDATE CASCADE,
    employee_id     INT NOT NULL REFERENCES users(user_id) ON UPDATE CASCADE,
    quantidade      INT NOT NULL CHECK (quantidade > 0),
    preco_unit      NUMERIC(8,2) NOT NULL CHECK (preco_unit >= 0),
    line_total      NUMERIC(10,2) GENERATED ALWAYS AS (quantidade * preco_unit) STORED,
);
```

Figure 96 - Openconsumption table in Database

This solution ensures that the total value of the line always remains consistent with the quantity and unit price, centralizing the calculation logic at the persistence level and avoiding discrepancies between the application and the database.

The opencard table also implements a privacy restriction through a CHECK constraint, requiring the removal of the phone number every time the card is deactivated (status = 'D'), contributing to compliance with data protection principles.

6.4.3.2 Security: Roles and Views

Access to data is not uniform; it depends on the role of the user authenticated in the database.

- **Restricted views:** Users with the EMPLOYEE role do not have direct access to the base tables. Access is exclusively through the vw_func_opencard view, which filters inactive cards and hides sensitive fields, such as the phone number.
- **System roles:** The startup script explicitly creates the owner_role (full access) and employee_role (read-only views) roles, ensuring that even in the event of an application failure, an employee cannot change or delete consumption history.

6.4.3.3 Abstraction Layer in C++ (Database.cpp)

The application adopts a clear separation of responsibilities, avoiding the dispersion of SQL commands throughout the code. All interaction with the database is encapsulated in the Database class, which implements the DAO (Data Access Object) pattern on top of the native libpq library.

Data Base Connection

The Database class explicitly manages the lifecycle of the TCP connection to the PostgreSQL server. The connect () method implements resilient reconnection logic: if it detects that the conn_ pointer exists but the connection has been lost, memory is freed with PQfinish before establishing a new connection.

The class destructor ensures proper release of resources, preventing memory leaks when the application closes.

```
15 DbResult Database::connect() noexcept {
16     try {
17         if (conn_ != nullptr) {
18             PGconn* pg = static_cast<PGconn*>(conn_);
19             // Check if existing connection is still healthy
20             if (PQstatus(pg) == CONNECTION_OK) {
21                 return DbResult::Ok;
22             }
23             PQfinish(pg);
24             conn_ = nullptr;
25         }
26
27         PGconn* pg = PQconnectdb(connString_.c_str());
28
29         if (PQstatus(pg) != CONNECTION_OK) {
30             PQfinish(pg);
31             return DbResult::ConnectionError;
32         }
33
34         conn_ = static_cast<void*>(pg);
35         return DbResult::Ok;
36
37     } catch (...) {
38         return DbResult::UnknownError;
39     }
40 }
```

Figure 97 - Function connect(), database.cpp

6.4.3.4 Fluxo de Integração: Do SQL ao Hardware

The integration between Database.cpp and the other modules is crucial for the deterministic behavior of NexiPass. The flow of a purchase operation illustrates this interaction:

1. **Event:** CardService receives a consumption request.
2. **Transaction in C++:** The db->registerConsumption(...) method is invoked.

3. **SQL execution:** The INSERT query is constructed and sent via PQexec.

4. **Server validation:** PostgreSQL automatically checks the constraints:

    - Existence of the card in the opencard table.
    - Quantity validation (CHECK qty > 0).

5. **Physical feedback:**

    - If successful, the method returns DbResult::Ok and CardService activates the green LED.
    - In case of failure (example: constraint violation), returns DbResult::ConstraintFai and the red LED is activated.

In this way, the visual feedback provided to the user truly reflects the state of the database.

6.5. Software Architecture and Concurrency

To meet latency and responsiveness requirements, Nexipass adopts a multithreaded architecture based on the Producer-Consumer pattern, designed to decouple data acquisition from business processing.

6.5.1. Concurrency Model

The ApiController class serves as the system manager. It is responsible for initializing three parallel threads, each with isolated responsibilities to prevent cascading blockages:

```
43 void ApiController::start() {
44     if (running_) return;
45     running_ = true;
46
47     thNFC_ = thread(&ApiController::nfcThreadFunction, this);
48     thWorker_ = thread(&ApiController::workerThreadFunction, this);
49     thNetwork_ = thread(&ApiController::networkThreadFunction, this);
50
51     // priorities: 1 (lowest) to 99 (highest)
52     setThreadPriority(thNFC_.native_handle(), 80);      // High priority
53     setThreadPriority(thNetwork_.native_handle(), 50);  // Medium priority
54     setThreadPriority(thWorker_.native_handle(), 30);   // Low priority
55
56 }
```

Figure 98 - Creating and assigning priorities to threads

1. **NFC Thread (**thNFC**) :**

- **Priority**: High (Soft Real-time).

- **Responsibility**: Performs a continuous polling cycle on the hardware responsible for RFID readings. As soon as a UID is detected, it is placed in a queue (workQueue) and the thread immediately returns to listening, minimizing inactive reading time.

```
55  void ApiController::nfcThreadFunction() {
56      SimpleRFID rfid;
57      if (!rfid.isReady()) {
58          cerr << "[NFC] Hardware SPI not ready.\n";
59          return;
60      }
61
62      string lastRawUid = "";
63
64      while (running_) {
65          if (rfid.isCardPresent()) {
66              string uid;
67              if (rfid.readCardUID(uid)) {
68                  if (uid != lastRawUid) {
69                      {
70                          lock_guard<mutex> lock(queueMtx_);
71                          workQueue_.push(uid);
72                      }
73                      queueCv_.notify_one();
74                      lastRawUid = uid;
75                  }
76              }
77          } else {
78              lastRawUid = "";
79          }
80
81          this_thread::yield();
82      }
83  }
```

Figure 99 - High-priority NFC thread

2. Worker Thread (thWorker**):**

- **Priority:** Low (Soft Real-time).
- **Responsibility:** Consumes the UIDs read for the queue and executes the business logic. It is this thread that establishes the connection with the PostgreSQL Database through the functions available in CardService, validates jumps, and decides if access is authorized.

- Updates the shared structure latestCardState, allowing the API to query the status of the last card without re-querying the database.

```cpp
void ApiController::workerThreadFunction() {
    while (running_) {
        string uidToProcess;

        {
            unique_lock<mutex> lock(queueMtx_);
            queueCv_.wait(lock, [this] {
                return !workQueue_.empty() || !running_;
            });

            if (!running_ && workQueue_.empty()) break;

            uidToProcess = workQueue_.front();
            workQueue_.pop();
        }

        CardSummaryDTO summary;
        db_->setEmployeeViewMode(false);
        DbResult res = db_->getCardSummary(uidToProcess, summary);

        CachedCardState newState;
        newState.card_id = uidToProcess;
        newState.scan_time = chrono::steady_clock::now();

        if (res == DbResult::Ok) {
            newState.is_valid_in_db = true;
            newState.status = summary.status;
            newState.total_pay = summary.total_to_pay;
        } else {
            newState.is_valid_in_db = false;
        }

        {
            lock_guard<mutex> lock(stateMtx_);
            latestCardState_ = newState;
        }

        cout << "[Worker] Processado: " << uidToProcess << " (Válido: " << newState.is_valid_in_db << ")\n";
    }
}
```

Figure 100 - low-priority worker thread

3. **Network Thread (**thNetwork**):**

- **Priority:** Medium (Soft Real-time).
- **Responsibility**: Keeps the HTTP server (httplib) active. Isolating this thread ensures that even if a network client has a slow connection, the physical reading of cards at the port is never affected.

```cpp
void ApiController::networkThreadFunction() {
    server_.set_default_headers({
        {"Access-Control-Allow-Origin", "*"},
        {"Access-Control-Allow-Methods", "POST, GET, OPTIONS"},
        {"Access-Control-Allow-Headers", "Content-Type"}
    });
    server_.Options(".*", [](const auto&, auto& res) { res.status = 204; });

    // LOGIN
    server_.Post("/login", [this](const auto& req, auto& res) {
        try {
            auto body = json::parse(req.body);
            UserDTO user;
            if (db_->authenticateUser(body["username"], body["password"], user) == DbResult::Ok) {
                json j; j["ok"] = true; j["role"] = user.role;
                j["user_id"] = user.user_id;
                db_->setEmployeeViewMode(user.role != "OWNER");
                res.set_content(j.dump(), "application/json");
            } else {
                res.set_content("{\"ok\":false}", "application/json");
            }
        } catch(...) { res.status = 400; }
    });
```

Figure 101 - Medium-priority network thread

6.5.2. Synchronization and Security (Thread Safety)

Sharing data between threads required the use of strict concurrency control mechanisms:

Mutual Exclusion (std::mutex): Access to message queues and global state variables is protected by mutexes, preventing race conditions where two threads would attempt to compete for the same resource simultaneously.

Condition variables (std::condition_variable): Used to notify the worker thread that there is already a uid in the queue and there is work to be done, preventing this thread from consuming CPU in busy waiting.

6.5.3. Signal management

To ensure safe shutdown, traditional Signal Handlers have been abandoned. The system uses signalfd to convert SIGINT (Ctrl+C) and SIGTERM signals into file descriptors. This allows the main loop to receive the stop request synchronously and in an orderly manner, ensuring that no resources are left hanging or corrupted when shutting down.

6.6. Business logic and external interface (API)

The upper layer of the System is responsible for transforming physical events, such as card readings, into useful functions for the system, such as registering products and making data available to external clients (mobile app).

6.6.1. Service Layer and Persistence

Interaction with the PostgreSQL database is not done on an ad-hoc basis. The **Data Access Object (DAO)** standard has been implemented in the Database class, which encapsulates all the complexity of the native libpq library.

The CardService class acts as a business logic layer. Before recording a transaction, it validates preconditions, ensuring that the validation logic is independent of SQL.

```
30  /* ================== ADD CONSUMPTION ================== */
31  DbResult CardService::addConsumption(const string& nfc_uid, int32_t productId, int32_t employeeId, int32_t quantity) {
32      if (quantity <= 0 || productId <= 0 || employeeId <= 0) {
33          feedback_->errorFB();
34          return DbResult::ConstraintFailed;
35      }
36
37      DbResult result = db_->registerConsumption(nfc_uid, productId, employeeId, quantity);
38
39      if (result == DbResult::Ok) {
40          feedback_->activateFB();
41      } else {
42          feedback_->errorFB();
43      }
44
45      return result;
46  }
```

Figure 102 - Pre-condition validation in CardService before database transaction

6.6.2. API REST

The ApiController exposes an HTTP JSON interface through the httplib library. Given the performance requirements in the application's GUI, a memory state cache strategy was implemented. This prevents the excessive load that the GUI would generate if it constantly polled to find out if the card had been read. To solve this problem, CachedCardState was used. When the Worker Thread processes a card, it updates latestCardState_ protected by a mutex. Then, the GET /scan_status endpoint reads only this memory structure. This allows for an instant response to the GUI.

To enable remote monitoring and interaction with the graphical interface, the system exposes a RESTful API on port 5000. The implementation uses the cpp-httplib library for HTTP server management and json for data serialization.

Communication follows the JSON standard, ensuring compatibility with any frontend.

The following REST APIs have been implemented:

Method	Endpoint	Description
Post	/login	Operator/employee authentication at the POS terminal
Get	/wait_card	Polling mechanism used by the tablet to check if an RFID card has been detected by the reader.
Post	/activate_card	Start a new session, linking the physical UID of the card to the customer's mobile phone number.
Post	/add_consumption	Record a purchase transaction, debiting the product amount from the customer's virtual balance.
Post	/validate_exit	Verify that the customer's balance is paid before authorizing the checkout.
Get	/products	Provides the updated list of products (Catalog) to fill in the menu of the graphical interface.
Get	/card_summary	View details of the current session: accumulated balance and list of products consumed.
Post	/close_card	Check out, ending the session and releasing the card for a new user.
Get	/product_totals	Generates the sales report aggregated by employee for viewing on the Web Dashboard.

Table 8 - Rest endpoint

6.7. Human-Machine Interface

User interaction with the NexiPass ecosystem is carried out via a mobile application developed in Flutter. Unlike a standard commercial application, this interface acts as a direct extension of the embedded system, requiring precise synchronization with hardware threads (NFC) and with the security rules defined in the database.

6.7.1. Architecture and Backend Integration

The mobile application architecture mirrors the structure of the previously developed C++ server. It follows the simplified **MVVM (Model-View-ViewModel)** pattern, where the communication logic is decoupled from the graphical interface.

Singleton ApiController "Bridge"

The core of the application resides in the ApiController class (in api_controller.dart). This class has been implemented as a Singleton, ensuring that session state, such as the authentication token, server IP, and user role, is maintained consistently throughout the navigation tree.

Communication with the REST C++ API uses strict JSON serialization. The data structures in Dart are designed to map the DTOs (Data Transfer Objects) defined in Database.h.

```
604  class ProductTotal {
605    final String productName;
606    final String employeeName;
607    final int totalQuantity;
608    final double totalRevenue;
609
610    ProductTotal({
611      required this.productName,
612      required this.employeeName,
613      required this.totalQuantity,
614      required this.totalRevenue,
615    });
616
617    factory ProductTotal.fromJson(Map<String, dynamic> j) {
618      return ProductTotal(
619        productName: j['product_name'] as String,
620        employeeName: j['employee_name'] as String,
621        totalQuantity: j['total_quantity'] as int,
622        totalRevenue: (j['total_revenue'] as num).toDouble(),
623      );
624    }
625  }
```

Figure 103 - Flutter singleton ApiController

This approach eliminates data interpretation errors between the Linux server (backend) and the mobile client.

6.7.2. Asynchronous Synchronization (Physical Event Polling)

One of the critical challenges of distributed systems without WebSockets is the detection of physical events initiated by the server. In NexiPass, RFID card reading occurs in the Raspberry Pi's thNFC thread, but the mobile application needs to react instantly to this event.

Implementation of Non-Blocking Polling

To solve this, an intelligent polling pattern was implemented. Unlike a traditional infinite loop that would block the UI, the application uses async/await primitives and the Flutter Widgets lifecycle.

In the entry_page.dart file, the _waitForCard method keeps an active poll to the /wait_card endpoint only while the screen is visible (mounted):

```dart
Future<void> _waitForCard() async {
  setState(() {
    _statusText = 'Waiting for NFC connection';
    _cardId = null;
  });

  while (mounted) {
    final String? cardId = await api.waitForCard();

    if (!mounted) return;

    if (cardId != null) {
      setState(() {
        _cardId = cardId;
        _statusText = 'Cartão lido: $cardId';
      });
      return;
    }


    await Future.delayed(const Duration(seconds: 1));
  }
}
```

Figure 104 - Non-blocking UI polling

This logic complements the Soft Real-Time architecture of the backend: while the C++ thread polls the SPI hardware, the App polls the network, balancing latency and power consumption.

6.7.3. End-to-End Security (Role-Based Access Control)

The security implemented in the PostgreSQL database extends to the graphical interface. The system implements defense in depth, where the UI dynamically adapts to user permissions.

After login, the backend returns the user's role (OWNER or EMPLOYEE). The application uses this information to hide or display sensitive elements.

In the role_page.dart file, access to the financial report (ProductTotalsPage) is exclusive to administrators, as is the viewing of personal data at checkout:

Exemplo 5.3 - Adaptação da UI baseada em Roles (role_page.dart):

```
116          if (api.currentUserRole == "OWNER") ...[
117            const SizedBox(height: 16),
118            ElevatedButton(
119              onPressed: () {
120                Navigator.push(
121                  context,
122                  MaterialPageRoute(builder: (_) => const ProductTotalsPage()),
123                );
124              },
125              style: ElevatedButton.styleFrom(
126                minimumSize: const Size.fromHeight(50),
127                backgroundColor: Colors.blue,
128              ),
129              child: const Text('Vending\'s Totals'),
130            ), // ElevatedButton
```

Figure 105 - Access control in frontend: Conditional widget rendering based on currentUserRole

This ensures that even if an employee attempts to access a restricted feature, the interface will not allow it, and the database would reject the query if the API were bypassed.

6.7.4. Dynamic Configuration and Usability

To ensure applicability in a real industrial environment, the system does not depend on *hardcoded* static IP addresses.

Persistence of Settings:

The settings_page.dart file uses the shared_preferences plugin to store the connection IP history. This allows the POS terminal to restart and automatically reconnect to the Raspberry Pi without technical intervention, increasing the robustness of the solution.

Data Validation at Source:

To reduce server load and improve user experience, data validation is performed preventively on the client side. On the activation page (activate_card_page.dart), Regular Expressions (Regex) are used to ensure data compliance before submission:

```
23      final text = _phoneController.text.trim();
24      final valid = RegExp(r'^\d{9}$').hasMatch(text);
25
26      if (!valid) {
27        _showSnack('Número de telefone inválido (use 9 dígitos).');
28        return;
29      }
30
```

Figure 106 - Data validation in Frontend

6.7.5. Implementation Summary

The mobile application completes the NexiPass system, transforming electrical signals and backend bits into an intuitive management tool. Integration was achieved through three main vectors:

1. **Data:** Strict JSON/DTO type mapping.
2. **Time:** Synchronization by asynchronous polling.
3. **Security:** Propagation of SQL roles to visual widgets.

7. Tests and system validation

NexiPass validation followed a methodology of Integration Testing and System Testing (Black-box testing). The objective of this phase was not only to verify isolated functions, but also to ensure that the orchestration between the three pillars of the project: Hardware (Raspberry Pi/RFID); Backend (C++/PostgreSQL); and Frontend (Flutter App); meets the requirements of determinism and security initially defined.

The tests were performed in a controlled environment, with the system powered up and connected to a dedicated local network. Four main aspects were evaluated:

1. **Functionality**: Is the business logic (purchases, balances, activation) correct?
2. **Synchronization**: Does the mobile app reflect the status of the hardware in real time?
3. **Security**: Do access rules prevent the viewing of sensitive data?
4. **Performance**: Does the system respond within Soft Real-Time limits?

7.1. General System Testing Matrix

Table X presents the end-to-end test scenarios performed, comparing the expected behavior with the actual behavior observed in the final prototype.

#	Test Scenario	Expected Result	Actual Result	Status
1	ColdBoot (Buildroot): Turn on the Raspberry Pi and wait for the service to be ready	The system should be operational in less than 10 seconds.	The system boots up and stabilizes in a few seconds. The NFC thread is successfully started.	Pass
2	App-Server Connectivity: Configure IP in the Flutter App and attempt connection.	The app should validate the IP, connect to APIREST, and display the login screen.	Connection established via HTTP. SettingsPage saves the IP in SharedPreferences for future use.	Pass

3	RFID reading (Latency): Bring a card close to the reader.	Immediate physical feedback: Buzzer sounds and LED turns green (if active) or red (if inactive).	Imperceptible latency, the SPI driver detects reading and acts on the GPIOs instantly.	Pass
4	Purchase with Balance: Record product consumption on a card with balance.	Transaction accepted in DB. Virtual balance updated and green LED flashes.	Record inserted in the openconsumption table. Colunalinetotal calculated correctly by Postgres.	Pass
5	Restriction Violation: Attempting to purchase a product with an invalid card.	The DB should reject the insertion (ConstraintFailed). Red LED and error sound activated.	Database.cpp captures the SQL error. Red visual feedback is triggered, and the app displays an error.	Pass
6	Security (Role Employee): Log in with user Role:EMPLOYEE and attempt to access totals.	Hidden “Vendings Totals” button. Customer's mobile phone data hidden at checkout.	The app hides sensitive widgets based on the currentUserRole flag. Restricted access confirmed.	Pass
7	Persistence and Integrity: Shutting down the system abruptly after a purchase.	When restarting, the card balance and sales history remain correct.	PostgreSQL guarantees durability. No data is lost after restarting.	Pass

Table 9 - Tests and system validation

7.2. Analysis of Results

The tests demonstrate that the system meets all critical functional and non-functional requirements. The robustness of the data layer is highlighted: the decision to implement constraints (‘CHECK Constraints’) and calculations ('GENERATED COLUMNS’) directly in the PostgreSQL engine prevented, in Test #6, the application from entering an inconsistent state, even when forced.

In terms of usability, the delay introduced by mobile application polling (Test #4) proved to be acceptable for the use case of a POS terminal, not compromising the fluidity of the operation,

while the physical feedback (LED/Buzzer) remains in strict real time, ensuring user confidence in the interaction with the hardware.

8. Conclusion

The implementation of NexiPass resulted in a hybrid system that combines the performance of direct hardware access (via Driver Kernel and direct SPI) with the flexibility of modern C++ (REST API and JSON). The use of advanced Linux primitives (signalfd, timerfd, epoll) allowed us to create a multithreaded, lock-free solution suitable for continuous operation in an industrial environment.

9. Gantt chart

The Gantt chart shown in Figure 107 reflects a balanced distribution of the project phases, ensuring consistency between planning, development, and final delivery. The Project Plan phase ran until 29/09, as stipulated, bringing together the definition of the problem, requirements, constraints, and overview of the system.

This will be followed by the Analysis phase, which will be short but sufficient to gather market information and outline the system and hardware architecture. Subsequently, the Design phase will take approximately three weeks, allowing for the specification of hardware and software, the selection of components, and preliminary testing.

The Implementation phase will be the most extensive, organised in parallel between software and hardware to optimise time, culminating in a period dedicated exclusively to integration testing and system validation.

Finally, documentation will be developed throughout the project, accompanying the analysis, design and implementation phases, to avoid overload at the end, and concluding with the preparation of the final report. This planning ensures not only that deadlines are met, but also that there are safety margins that allow for dealing with possible unforeseen events without compromising the quality of the project.

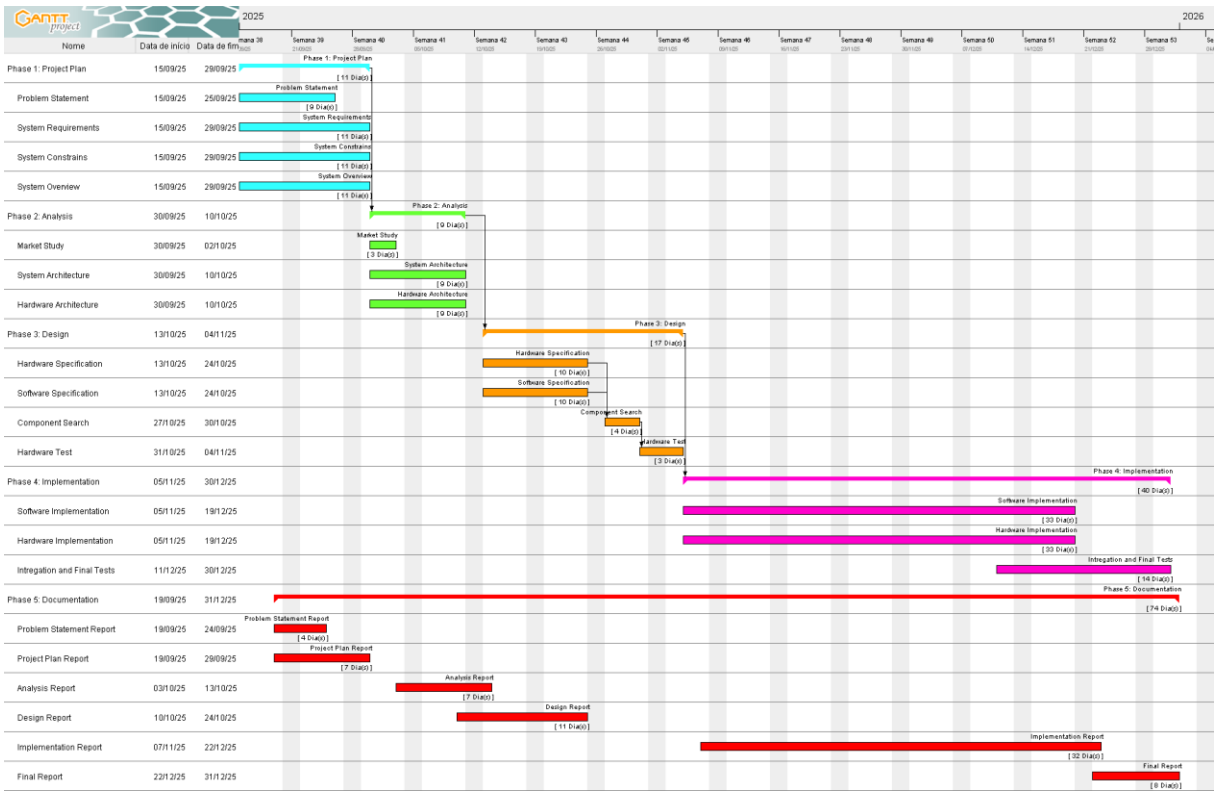

Figure 107 - Gantt Chart