



---

# TALLER DE PROYECTO II

## INFORME FINAL

---

**G3 - Simulador de ATM con Raspberry Pi y tablero de control web**



### **GRUPO DE DESARROLLO**

- BLANCO VALENTIN NICOLAS – 02326/5
- BONIFACIO LUCAS GABRIEL – 01982/0
- CALDERON SERGIO LEANDRO – 02285/4
- PALADINO GABRIEL AGUSTÍN – 02490/8

# Índice

1 – Introducción.....	1
2 – Objetivos .....	2
2.1 – Objetivos primarios .....	2
2.2 – Objetivos secundarios.....	3
2.3 – Consideraciones .....	3
3 – Funcionalidades .....	4
3.1 – Cajero ATM para clientes .....	4
3.2 – Tablero web para administrador.....	5
3.3 – Conexionado físico .....	6
3.4 – Procesos .....	8
4 – Estructura del proyecto.....	13
5 – Instalación .....	14
5.1 – Mosquitto .....	14
5.2 – Entorno de Python .....	15
6 – Sistema back-office .....	16
6.1 – Backend.....	16
6.2 – Frontend .....	29
7 – Sistema cajero RPI.....	39
7.1 – Archivos Python .....	39
7.2 – Otros archivos.....	46
8 – Explicación de procesos.....	47
8.1 – Autenticación de cliente .....	47
8.2 – Consulta de saldo .....	49
8.3 – Ingreso de dinero .....	49
8.4 – Retiro de dinero .....	50

8.5 – Transferencia .....	51
8.6 – Actualización de límites.....	53
9 – Documentación relacionada .....	54
10 – Bibliografía .....	64
Apéndice .....	67
A.1 – Partes de hardware .....	67
A.2 – Materiales recibidos.....	67

## 1 – Introducción

Los cajeros automáticos o ATM (*Automated Teller Machines*), han transformado de manera significativa la manera en que las personas acceden y gestionan su dinero. Se introdujeron en la década de 1960 [1]; llegando a la ciudad de La Plata en el año 1986 [2]. Estas máquinas revolucionaron la industria financiera al ofrecer una amplia gama de servicios bancarios con acceso las 24 horas, por ejemplo: retiro de efectivo, consulta de saldo, transferencia de fondos y realización de transacciones. Su fuerte presencia en el país, con más de 18.000 terminales al año 2021, tanto en áreas urbanas e incluso rurales ha ampliado el alcance de estos servicios [3].

Las terminales pueden depender de una entidad bancaria, o bien, ser independientes, pudiendo ser instalados en comercios y puntos estratégicos para la circulación de las personas. En Argentina, Red Link ofrece la instalación, soporte técnico y monitoreo 24 horas de “Cajeros Express” con una inversión inicial de 7000 dólares (5 millones de pesos a septiembre 2023) [3].

El presente proyecto tiene como objetivo la simulación de un cajero ATM utilizando una Raspberry Pi, conectada a un servidor web que dispondrá de un tablero de control para el administrador. Empleando el microcontrolador mencionado y los periféricos descritos en este plan, se busca mostrar la viabilidad de la puesta en marcha de una terminal independiente a menor costo.



## 2 – Objetivos

La meta de este proyecto es construir una imitación de un cajero automático (ATM) mediante el uso de una Raspberry Pi como dispositivo central y un panel de control en línea para comunicarse con el ATM simulado. Para poder utilizar el mismo los usuarios se identificarán con una tarjeta RFID y el ingreso del PIN asociado, lo que le brindará acceso a un menú de opciones. Esta interfaz simple le dará la capacidad al usuario de llevar a cabo acciones bancarias ficticias, como revisar el saldo, añadir y extraer dinero, entre otras. Por otra parte, el administrador podrá auditar el ATM mediante el panel web.

### 2.1 – Objetivos primarios

#### 2.1.1 – Hardware cajero

- Lectura de tarjetas de débito y crédito mediante tecnología RFID para la identificación del usuario.
- Autenticación mediante PIN de 4 dígitos.
- Realizar un programa Python para la lógica de los estados.
- Realizar una aplicación de interfaz simple para las operaciones.
- Permitir la visualización del saldo actual en la cuenta.
- Poder realizar transferencias bancarias a partir de CBU.
- Simular la realización de depósitos y retiros bancarios.

#### 2.1.2 – Back office

- Poder consultar el estado del servicio del cajero ATM.
- Utilizar MongoDB como base de datos propia en el servidor web.
- Utilizar Express para el backend y React para el frontend. Al conjunto de estas aplicaciones se las denomina MERN (MongoDB + Express + React

+ NodeJS) y es lo que utilizaremos para el desarrollo del tablero de control web exclusivo del administrador [4].

- Registrar y listar usuarios, tarjetas, transferencias y cuentas.
- Banear y desbloquear el uso de ciertas tarjetas.
- Conocer y actualizar los límites de retiro en el cajero.

## **2.2 – Objetivos secundarios**

- Bloqueo de tarjeta ante sucesivos intentos fallidos de autenticación
- Ofuscación del PIN de la tarjeta en la comunicación entre partes.
- Registro detallado de los depósitos y extracciones realizadas.
- Generación de informes según las transacciones en un rango de fechas.
- Diseño de un Home Banking para los usuarios clientes.
- Agregar dinero al cajero mediante el panel de backoffice

## **2.3 – Consideraciones**

Según las correcciones de la cátedra, los objetivos planteados son correctos, pero se realizó una separación entre “nivel de usuario” y “back-office”, y se agregaron algunos detalles adicionales para una mejor comprensión.

La posibilidad de realizar transferencias desde el cajero fue sugerido como objetivo secundario por la cátedra, pero debido a la disponibilidad de tiempo suficiente se lo consideró objetivo primario.

Por otra parte, para la generación de la interfaz del cajero en el microcontrolador se desarrolló primero una versión de consola, mostrado en el segundo informe de avance, pero finalmente fue reemplazada por una aplicación de Flask, accesible desde otros dispositivos a partir de la IP y puerto.

## 3 – Funcionalidades

### 3.1 – Cajero ATM para clientes

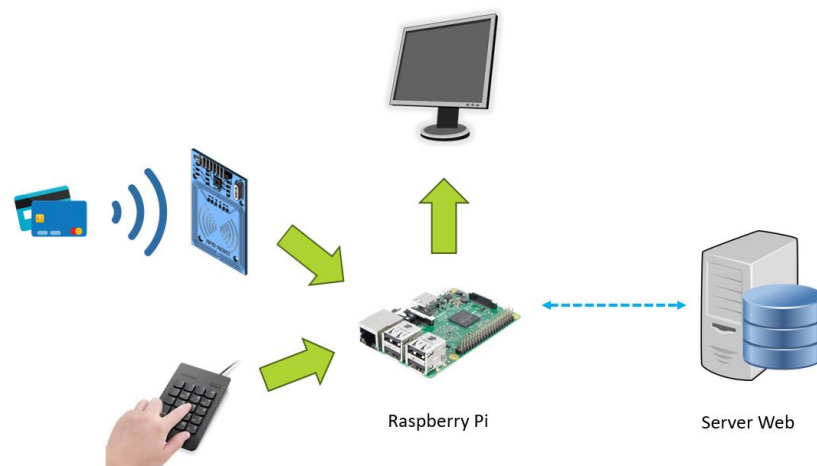
Implementando en Python en la Raspberry Pi. Se requiere de un monitor conectado a la salida HDMI del microcontrolador para la visualización de la consola, y de un teclado para escribir el comando que inicia el programa.

Una vez en ejecución, por consola se muestra la dirección IP y puerto que permite acceder al servidor con interfaz gráfica desde el navegador web de otro dispositivo, ya que la Raspberry Pi tiene instalado el SO Raspbian Lite.

La autenticación requiere 2 pasos:

- Acercar la tarjeta de débito o crédito al lector RFID Rc522.
- Ingresar el PIN asociado desde el navegador, por teclado.

Una vez ingresado en el sistema, se accede al menú de opciones para consulta de saldo, depositar dinero, retirar efectivo y realizar transferencia. Estas operaciones requieren comunicación MQTT con el servidor back-office, conectado a la misma red Wi-Fi, con una IP indicada al iniciar el programa.

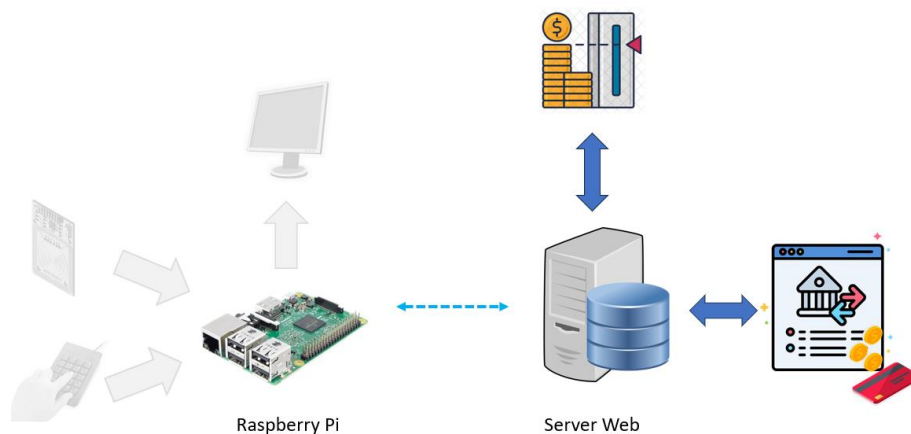


*Figura 3.1. Componentes comunicados con Raspberry Pi.*

### 3.2 – Tablero web para administrador

Implementado en una computadora aparte y desarrollado con el fullstack MERN. Está compuesto de un backend, conectado a la base de datos de MongoDB y a la Raspberry por comunicación MQTT; y, por otra parte, de un frontend, desde el cual se pueden realizar las siguientes operaciones:

- Verificar el estado de funcionamiento del cajero.
- Consultar el efectivo disponible en el cajero en tiempo real.
- Establecer límites para las extracciones.
- Dar de alta nuevos clientes y tarjetas a partir de formularios.
- Acceder a los listados de clientes, tarjetas, transferencias y cuentas.
- Conocer el número de tarjetas asociadas a cada cliente.
- Borrar clientes y tarjetas de manera permanente.
- Posibilidad de bloquear y reactivar tarjetas.



*Figura 3.2. Parte back-office del sistema*

A continuación, se muestra la integración de las dos partes mencionadas, resultando entonces los siguientes tipos de comunicación: HDMI entre Raspberry y monitor, USB entre Raspberry y teclado, protocolo SPI entre Raspberry y lector Rc522, RFID entre lector y tarjeta, MQTT entre Raspberry y backend, y HTTP entre backend y frontend del tablero, como también entre el servidor Flask del cajero y el navegador conectado.



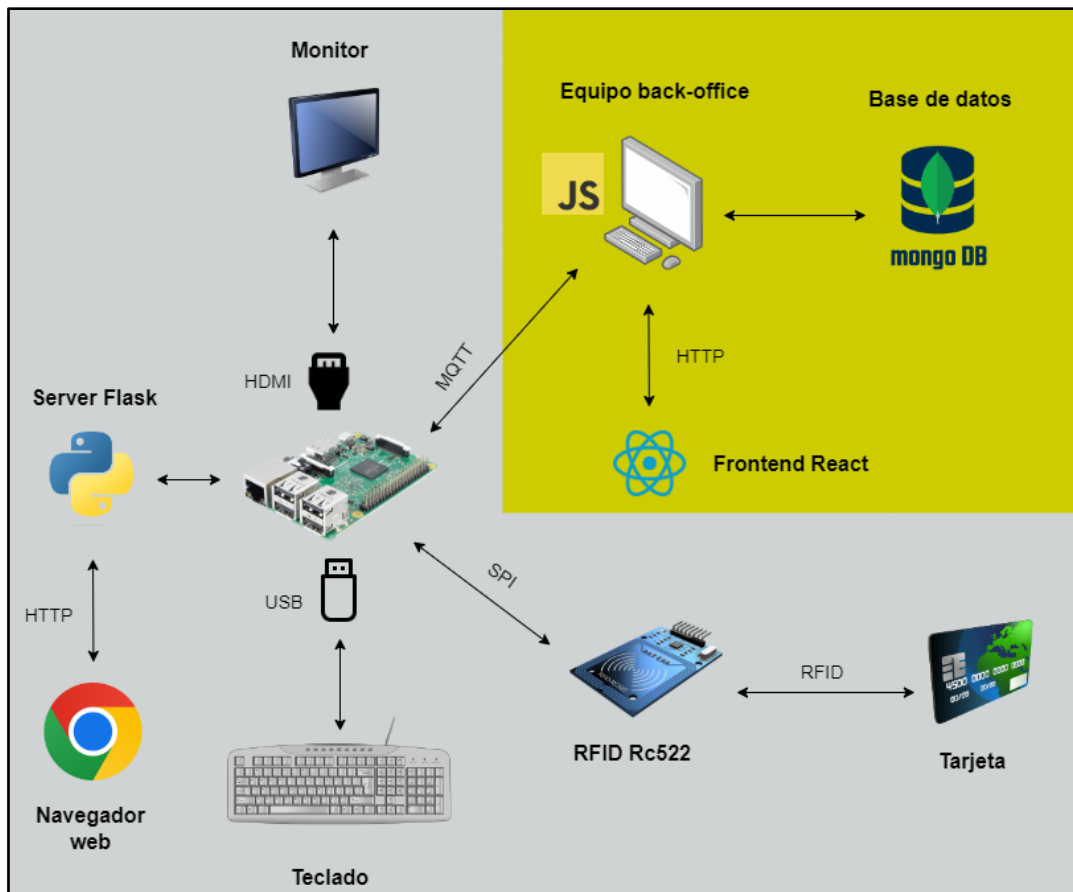


Figura 3.3. Comunicaciones del sistema completo.

### 3.3 – Conexión físico

El único componente conectado a la tira de pines del microcontrolador es el lector de tarjetas RFID, cuyo conexionado se muestra a continuación.

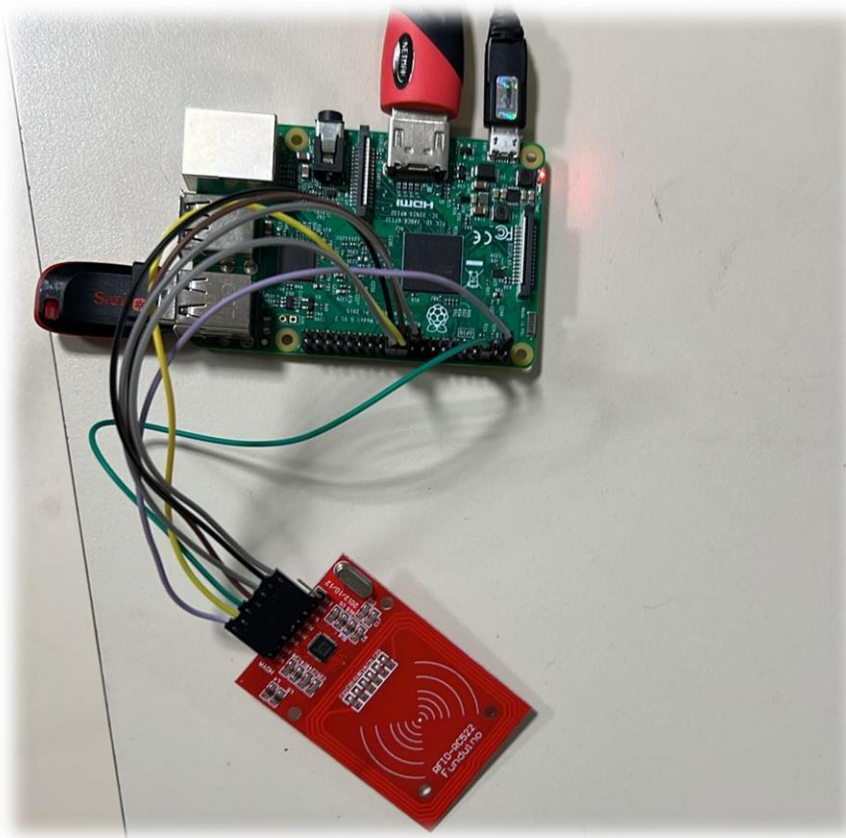


Figura 3.4. Esquemático del RFID Rc522 para Raspberry Pi.

El pin-out se corresponde con la siguiente tabla de conexiones.

Terminal RFID Rc522	Terminal Raspberry Pi	
	Número	Nombre
VCC	01	3.3V
RST	22	GPIO_GEN6
GND	06	Ground
MISO	21	SPI_MISO
MOSI	19	SPI_MOSI
SCK	23	SPI_CLK
NSS	24	SPI_CE0_N
IRQ	-	-

La siguiente imagen muestra la conexión de todos los periféricos al microcontrolador otorgado para el proyecto, además de su alimentación USB.

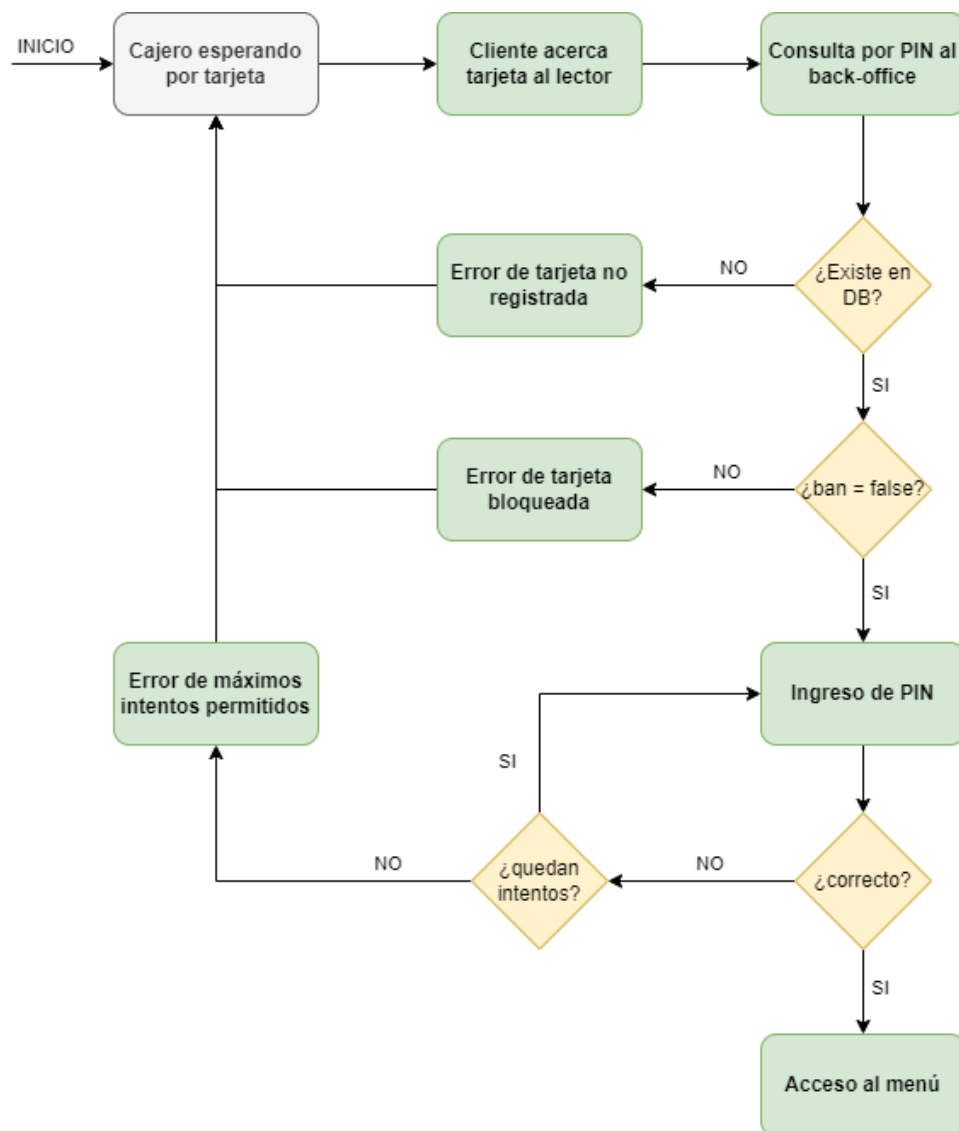


*Figura 3.5. Conexionado real de la Raspberry del proyecto.*

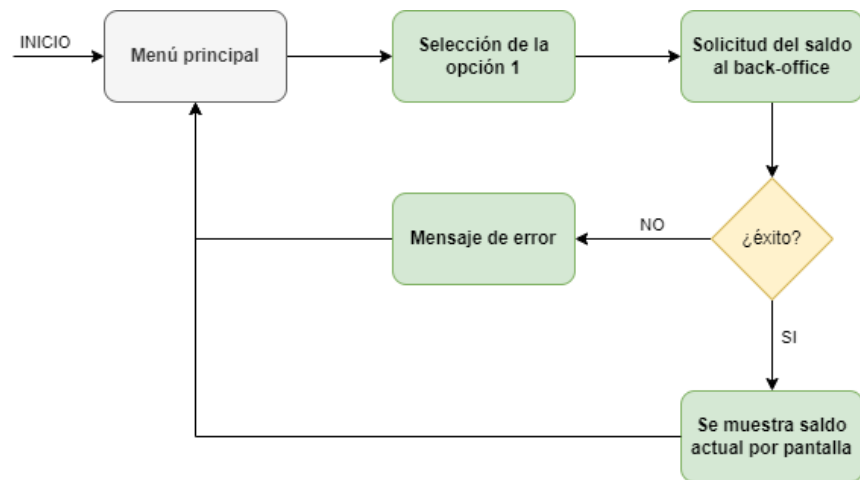
### 3.4 – Procesos

A continuación, se describen de manera esquemática los procesos principales que brinda el sistema con ambas partes integradas. Una descripción más detallada se encuentra en el capítulo 8, luego de presentarse los archivos.

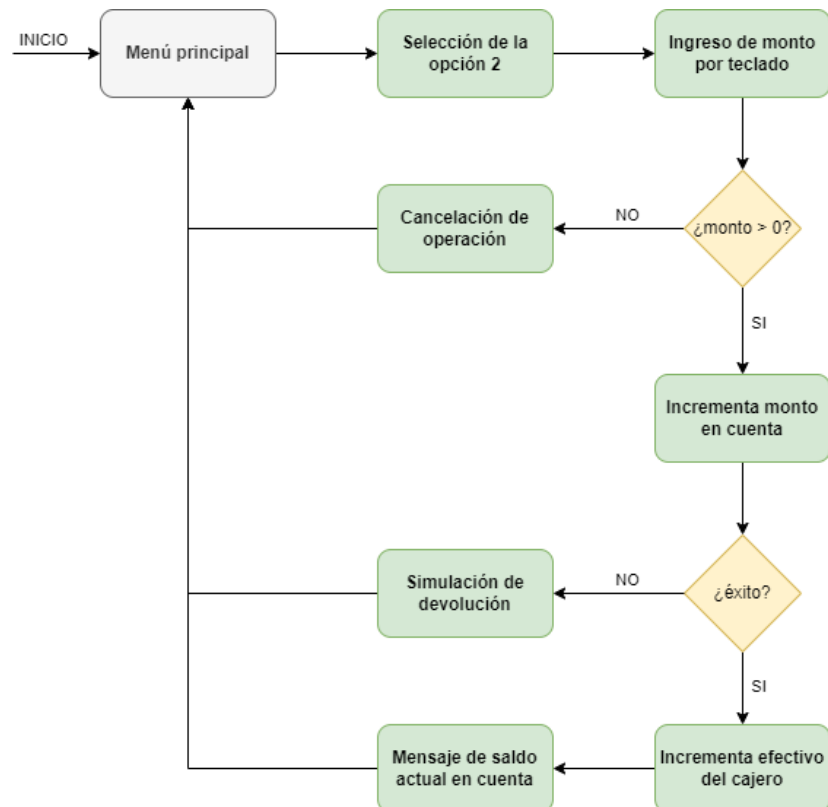
- ❖ **Autenticación del cliente en cajero:** consiste en la lectura de la tarjeta por parte del RFID, cuya implementación está realizada en la librería MFRC522 de Python. Posteriormente, se obtiene el PIN y se solicita el correcto ingreso de este para el acceso al menú de opciones.



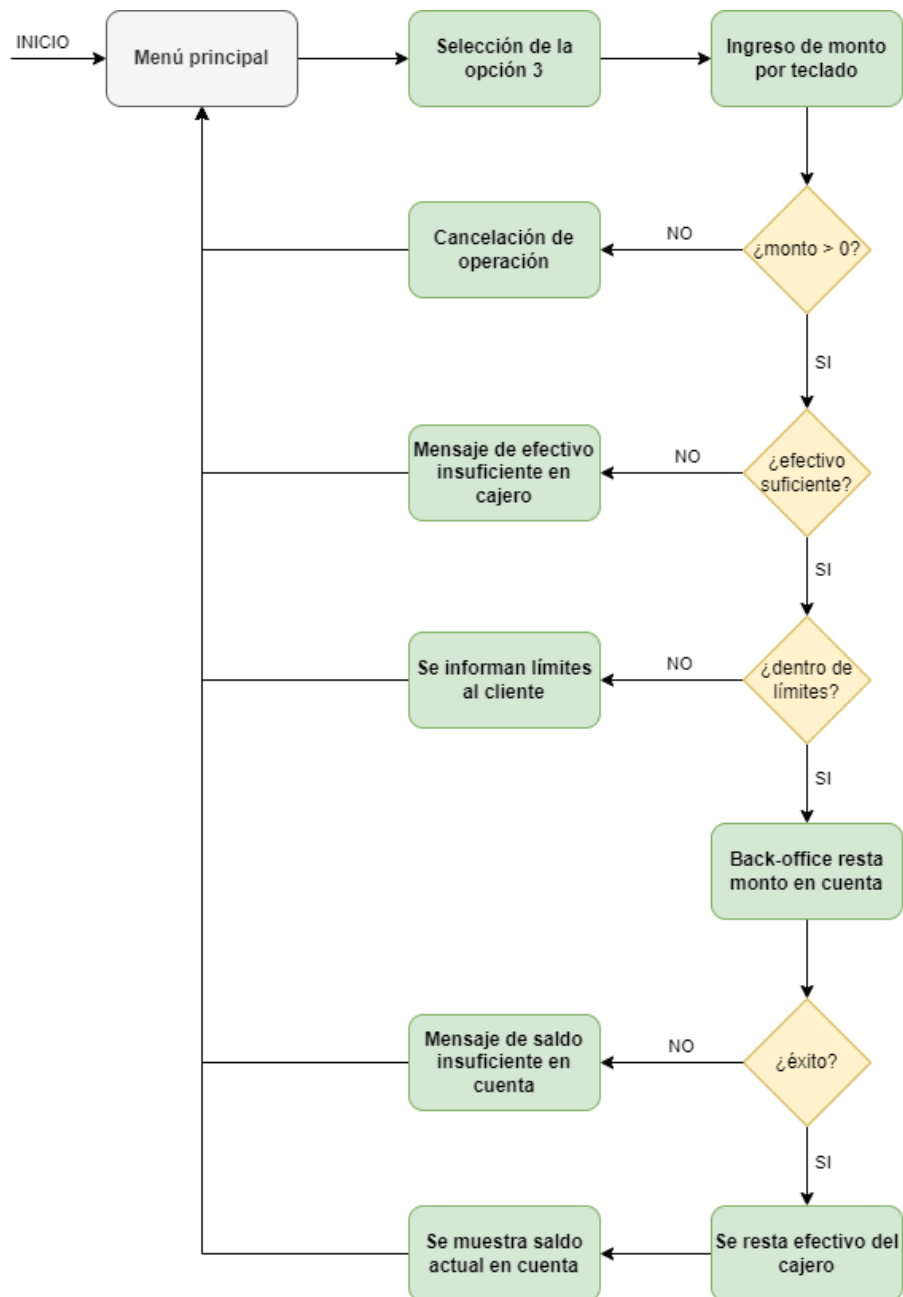
- ❖ **Consulta de saldo:** el cliente puede conocer el monto actual de su cuenta a partir del número de tarjeta ya autenticada.



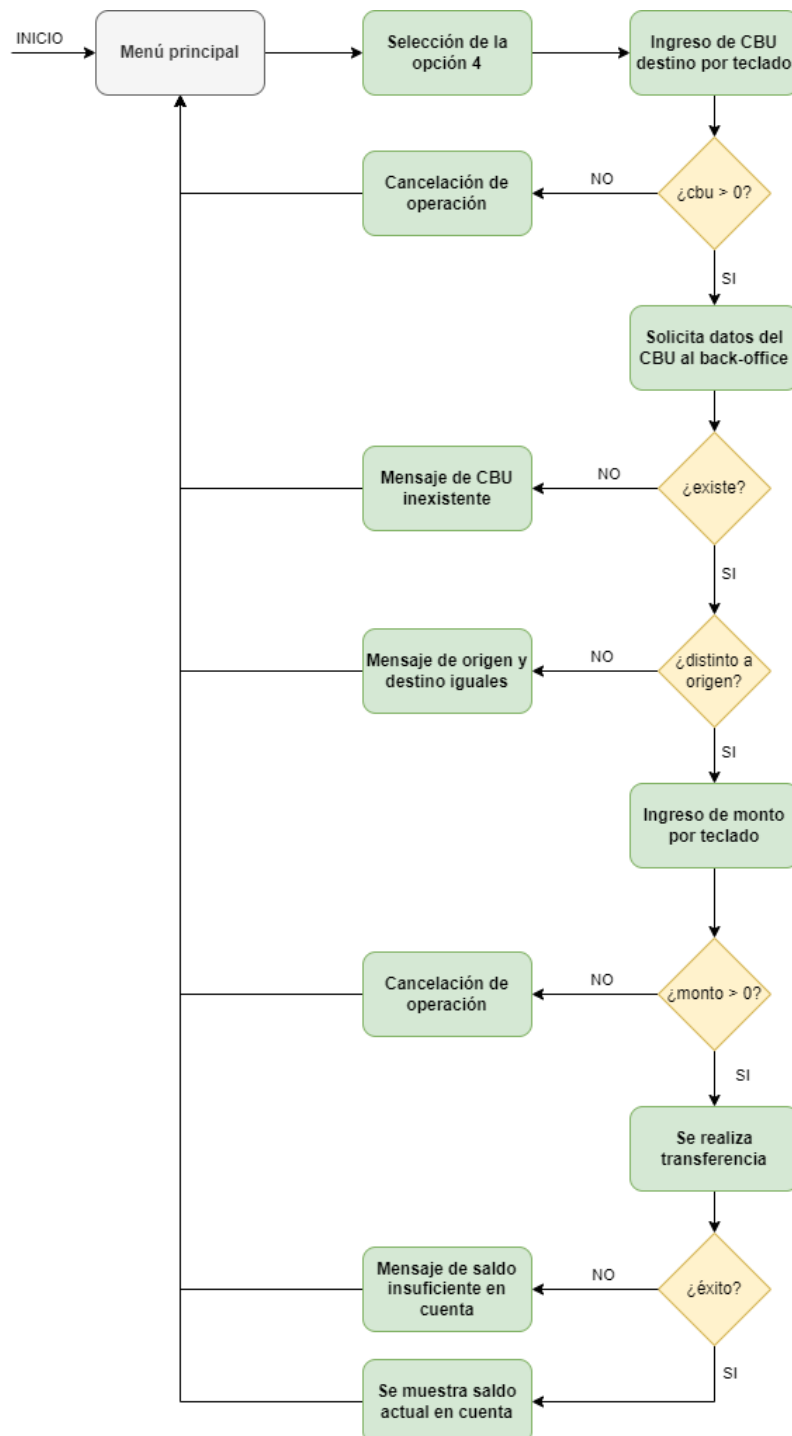
- ❖ **Ingreso de dinero en cuenta:** se simula el depósito de efectivo en el cajero por parte del cliente, solicitando el monto a adicionar en la cuenta.



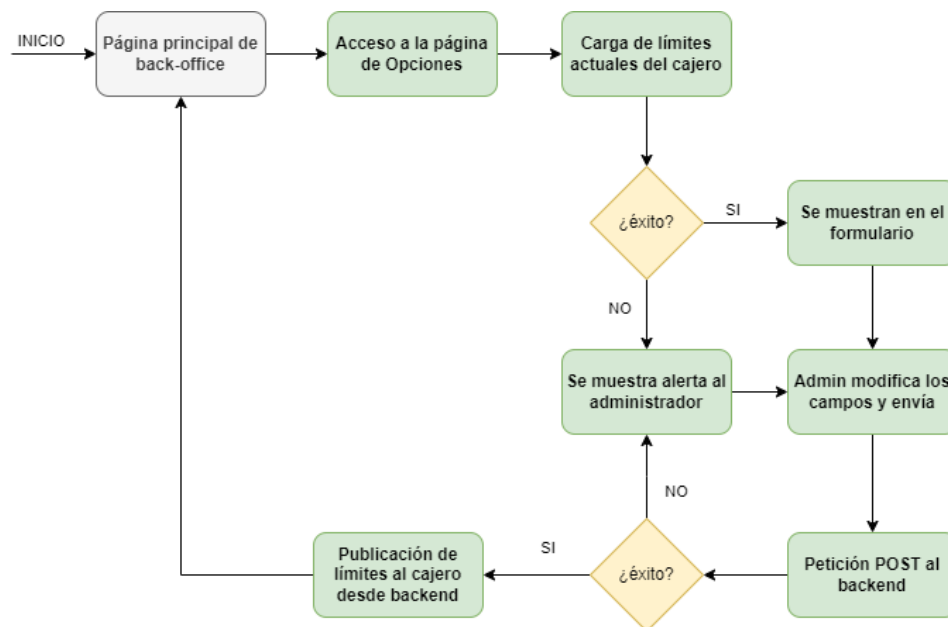
- ❖ **Retiro de dinero:** se simula la extracción de efectivo en el cajero por parte del cliente, solicitando el monto a adicionar en la cuenta, pero existe una cantidad mayor de controles respecto al depósito ya que además debe existir efectivo suficiente en el cajero, el monto debe encontrarse dentro de los límites permitidos y la cuenta debe tener un saldo mayor o igual a lo indicado por el usuario.



- ❖ **Transferencia:** desde el cajero, el cliente primero ingresa el CBU destino, el cual debe existir y ser distinto al de la cuenta propia. Luego, se solicita indicar el monto a transferir, realizándose la operación si no excede el saldo disponible en la cuenta, mostrando el monto final al completarse.



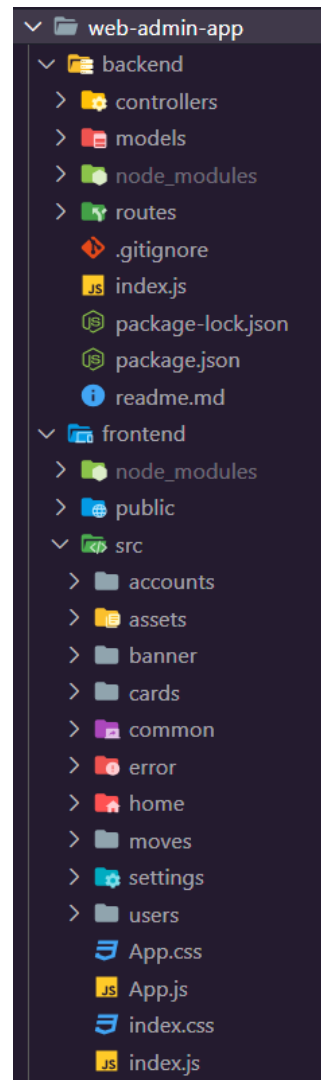
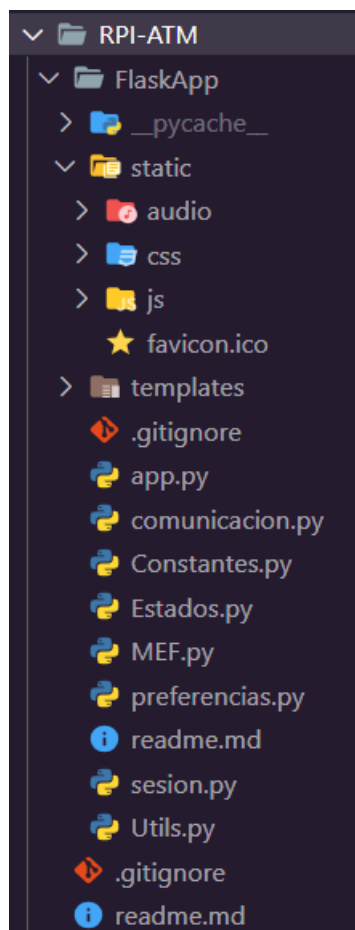
- ❖ **Actualización de límites de extracción:** disponible para el administrador en el panel web, donde puede consultar los límites actuales del cajero y modificarlos mediante un formulario simple.



## 4 – Estructura del proyecto

La organización de los archivos que conforman el proyecto en su totalidad se encuentra dividida en dos directorios principales:

- **RPI-ATM:** contiene los scripts de Python desarrollados para su ejecución en la Raspberry Pi 3, explicado en el capítulo 7 de este informe. Este es el único directorio que se requiere copiar al microcontrolador.
- **web-admin-app:** posee las dos partes del back-office (backend y frontend) que se ejecutan en la PC, explicadas en el capítulo 6.



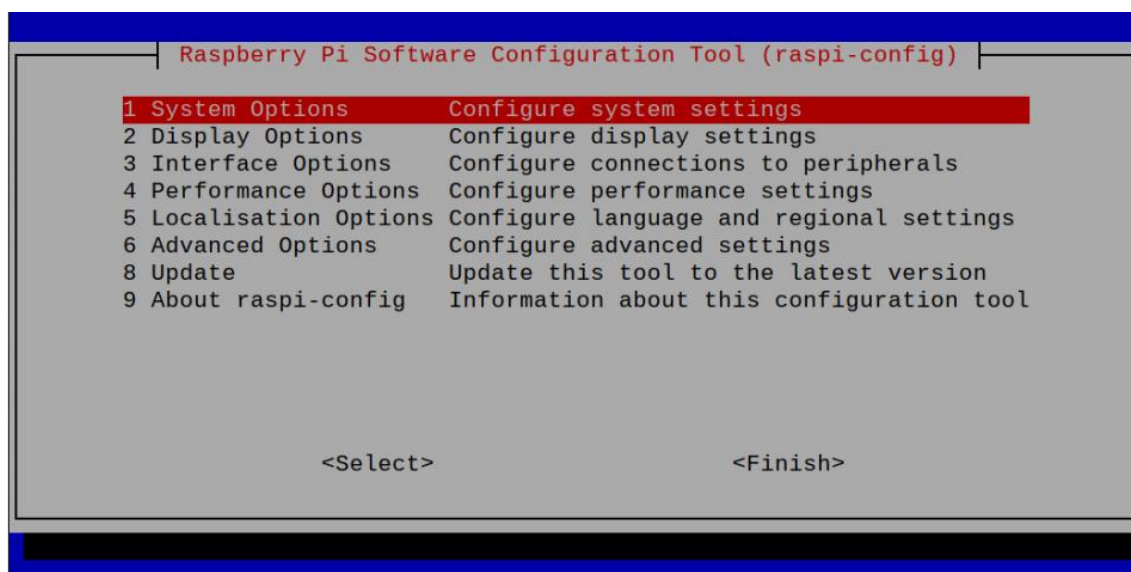
Figuras 4.1 y 4.2. Organización del sistema RPI y back-office.



## 5 – Instalación

El arranque del sistema operativo Raspbian se realiza mediante el USB brindado por la cátedra. La imagen también se puede descargar desde la página raspberrypi.com, eligiendo la versión Lite (sin escritorio).

La conexión a Internet se efectúa mediante el comando **sudo raspi-config**, y luego accediendo a “System Options” > “Wireless LAN”. Se ingresa el nombre de la red (SSID) y luego la contraseña. También es necesario activar SPI para la correcta comunicación con el lector RFID Rc522.



*Figura 5.1. Menú de la herramienta raspi-config*

Para la muestra, la conexión entre el tablero web y la Raspberry Pi se realiza mediante la red Wi-Fi de la facultad, permitiendo el intercambio de datos sin necesidad de cables Ethernet. Una consideración es que se debe verificar la conectividad con antelación, conociendo las direcciones IP de ambas partes.

### 5.1 – Mosquitto

Respecto a MQTT, se requiere instalar Mosquitto en ambas partes (mosquitto y mosquitto-clients en Raspbian) para así comunicarse con el bróker instalado en la PC. En un principio, la recepción no había sido exitosa.

Se investigó y se halló entonces que la recepción de los mensajes en la PC estaba siendo interceptada por el Firewall de Windows. La solución consiste en crear tanto una regla de entrada como de salida para el puerto **1883**, con la opción de “Permitir la conexión” habilitada en ambos escenarios [12].

Otra acción requerida para la correcta recepción de las publicaciones es la edición del archivo de configuración “mosquitto.conf” en la PC, estableciendo el parámetro “allow\_anonymous” en true [13]. Dicho archivo está disponible para su descarga directa en el directorio “broker” del repositorio GitHub del proyecto.

Una vez aplicadas las correcciones, se pueden realizar las publicaciones y suscripciones de tópicos MQTT entre la Raspberry Pi y la computadora con SO Windows, conociendo la dirección IP de esta última.

```
mosquitto_pub -h IP_WINDOWS -t “cajero/topico” -m “mensaje” -d
```

## 5.2 – Entorno de Python

En Raspbian, para la correcta ejecución del script del cajero es necesario crear un entorno virtual de Python, con el siguiente procedimiento [16].

- Crear entorno virtual de Python mediante: ``python3 -m venv .venv``
- Activar entorno virtual: ``source .venv/bin/activate``
- Actualizar PIP: ``python3 -m install --upgrade pip``
- Instalar Flask: ``python3 -m install Flask``
- Instalar librería MRFC522: ``python3 -m install mrfc522``
- Instalar librería Paho-MQTT: ``python3 -m install paho-mqtt``
- Ejecutar aplicación: ``python3 app.py [IP_WINDOWS]``

Al finalizar, se cierra el entorno virtual con el comando deactivate. En las próximas ejecuciones, es suficiente con activar el entorno virtual (paso 2) y ejecutar la aplicación pasando la IP correspondiente (paso 7).

## 6 – Sistema back-office

### 6.1 – Backend

Corresponde a la parte del sistema del administrador sin interfaz gráfica, cuyo programa principal es **index.js**, ejecutable con NodeJS mediante el comando “npm start” en el directorio backend.

Se utiliza el framework Express para levantar el servidor en un puerto indicado de localhost, en este caso, el número **2000**, y poder llevar a cabo el procesamiento de diferentes solicitudes de API Rest. En su inicialización, se deja explícito el uso del formato JSON en las peticiones y el módulo CORS (Cross-Origin Resource Sharing) de modo que los navegadores permitan al frontend solicitar recursos desde otros orígenes distintos al de la aplicación [6].

```
const express = require("express");  
  
const cors = require("cors");  
  
  
const app = express();  
  
app.use(express.json());  
  
app.use(cors());
```

*Código 6.1. Inicialización de Express*

Posee también conexión a la base de datos de MongoDB, realizado en la función **dbConfig()**, que a su vez invoca el método *connect()* de la librería Mongoose. Se especifica el puerto, por defecto **27017** de localhost, y el nombre “atm-db” de la base de datos, resultando “*mongodb://127.0.0.1:27017/atm-db*” como URL [7]. Los esquemas de documentos de cada colección deben estar representados en archivos JavaScript, los cuales se encuentran en la carpeta **models**, para luego crear los controladores y rutas (secciones 6.1.1 a 6.1.3)

### 6.1.1 – Modelos

Para cada colección, se creó un archivo que representa su esquema, es decir, los campos que posee cada documento y su tipo de dato. Se crearon un total de **4 colecciones**: tarjetas (card.js), cuentas (cuenta.js), transferencias (move.js) y clientes (user.js), mostradas en el modelo relacional de la Figura 6.1.

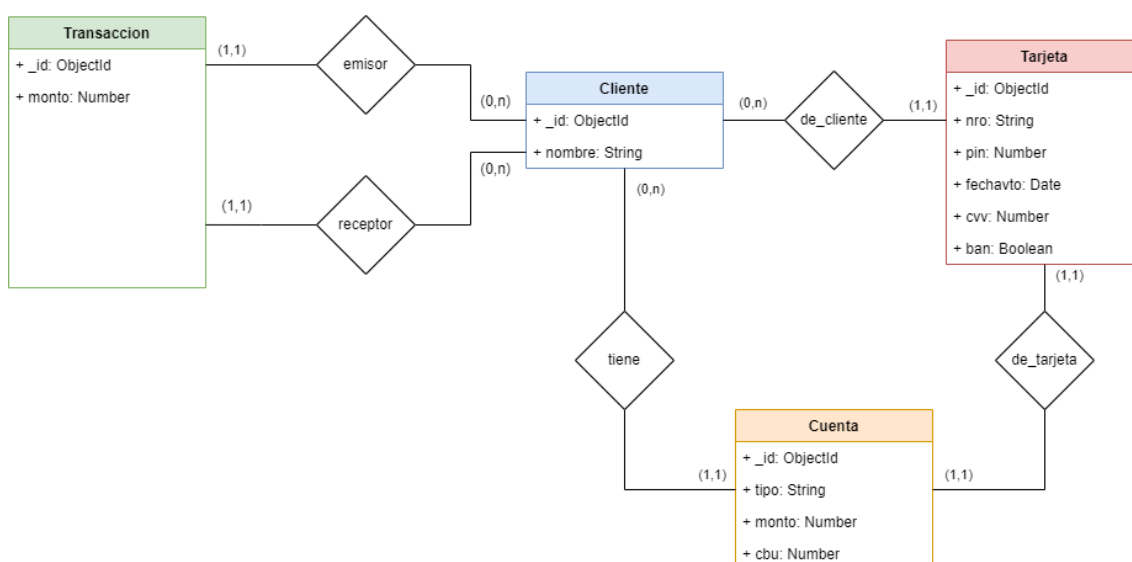


Figura 6.1. Colecciones de la base de datos “atm-db”

Cada documento posee un campo “\_id”, que es del tipo ObjectId y se autogenera al momento de inserción en la base de datos. Se tienen 3 modelos con referencias foráneas a un cliente particular, y cuentas siempre asociadas a tarjetas. A continuación, se detalla el significado de cada campo agregado.

- Modelo card.js

Campo	Descripción	Referencia
<b>cliente</b>	ID del cliente titular	_id de Cliente
<b>nro</b>	Número único al frente (12 a 16 dígitos)	-
<b>pin</b>	Clave de autenticación (4 dígitos)	-
<b>fechavto</b>	Fecha completa de caducidad	-
<b>cvv</b>	Código de seguridad al reverso (3 dígitos)	-
<b>ban</b>	Indicador de bloqueo	-

- Modelo cuenta.js

Campo	Descripción	Referencia
<b>cliente</b>	ID del cliente titular	_id de Cliente
<b>tarjeta</b>	ID de la única tarjeta asociada	_id de Tarjeta
<b>tipo</b>	Caja de ahorro o cuenta corriente	-
<b>monto</b>	Saldo disponible (mayor o igual a 0)	-
<b>cbu</b>	Clave bancaria uniforme (positivo)	-

- Modelo move.js

Campo	Descripción	Referencia
<b>emisorId</b>	ID de cliente origen (quien realiza)	_id de Cliente
<b>receptorId</b>	ID de cliente destino (quien recibe)	_id de Cliente
<b>monto</b>	Cantidad de dinero transferido (positivo)	-

- Modelo user.js

Campo	Descripción	Referencia
<b>nombre</b>	Nombre y apellido	-

Respecto al código, si en las consultas asociadas a un modelo se requiere realizar un “populate” (similar al join de SQL), es necesario indicar explícitamente el nombre de colección foránea. Por ejemplo, para las transferencias:

```
const schema = new mongoose.Schema({
  emisorId: {type: ObjectId, ref: "Cliente"},
  receptorId: {type: ObjectId, ref: "Cliente"},
  monto: Number
});
```

*Código 6.2. Esquema declarado en move.js.*

### 6.1.2 – Controladores

Una vez definidos los modelos, se desarrollaron las funciones de consulta, inserción, actualización y eliminación asociadas a cada uno, a lo que en su conjunto se denomina controlador. Se encuentran en el directorio **controllers**, y los archivos poseen el nombre del modelo acompañado del sufijo “Controller”.

En todos los casos, se definió la consulta “getCount” que devuelve la cantidad de documentos presentes en la colección correspondiente. Únicamente se realiza un llamado a la función `countDocuments()` del modelo.

También se implementaron funciones “getAll” para obtener todos los documentos con el método `find()` sin especificar parámetros. En el caso de los clientes, además se retornan las tarjetas asociadas en la misma consulta, mediante el método `aggregate().lookup()` como se muestra a continuación [8].

```
getAll: async(req, res) => {  
    users = await model.aggregate().lookup({  
        from: 'tarjetas',  
        localField: '_id',  
        foreignField: 'cliente',  
        as: 'tarjetas'  
    });  
    res.json({ Usuarios: users });  
}
```

*Código 6.3. Consulta para obtención de todos los clientes y sus tarjetas.*

Las dos funciones mencionadas son comunes para las 4 colecciones del proyecto. A continuación, se describe el resto de las operaciones disponibles acorde a cada modelo, y luego se explica cómo se realizan en código.

### Controlador cardController.js

- *postCard*: se encarga de cargar las tarjetas al sistema. Para esto recibe como parámetros el cliente, número de tarjeta (único), pin que tendrá asociado, fecha de vencimiento y cvv. Si los parámetros recibidos son correctos procede a crear y asociar una cuenta a la tarjeta.
- *getPin*: recibe como parámetro un número de tarjeta y la busca en el sistema. Si la encuentra y no está bloqueada, devuelve el PIN junto al ID que tiene en la base de datos, sino retorna un mensaje de error.
- *banearTarjeta*: recibe como parámetro el ID de una tarjeta. Si la encuentra, se bloquea poniendo en true el campo "ban" de la misma.
- *desbanearTarjeta*: recibe como parámetro el ID de una tarjeta. Si la encuentra, se desbloquea estableciendo el campo "ban" en false.
- *deleteCard*: recibe como parámetro el ID de una tarjeta. Si la encuentra, la borra de manera permanente junto a su cuenta asociada.

### Controlador cuentaController.js

- *getMonto*: recibe el ID de una tarjeta y si encuentra la cuenta asociada devuelve el monto.
- *ingresarMonto*: recibe como parámetro el ID de una tarjeta y el monto que se quiere adicionar a la cuenta que tiene asociada. Si encuentra la tarjeta suma el monto recibido como parámetro al monto que tiene la cuenta.
- *retirarMonto*: recibe como parámetro el ID de una tarjeta y el monto que se quiere retirar de la cuenta que tiene asociada. Si encuentra la tarjeta, verifica que el monto a retirar sea menor al monto que tiene la cuenta.
- *getCbuInfo*: a partir de un CBU, devuelve el nombre del cliente asociado o un mensaje de error si no existe o si coincide con la cuenta origen.

### Controlador moveController.js

- *postMove*: transfiere el monto indicado desde una cuenta origen hacia otra cuenta destino, indicada esta última por parámetro CBU. También se requiere pasar el número de la tarjeta origen para verificar que la transferencia no se esté realizando desde y hacia una misma persona.

### Controlador *UserController.js*

- *postUser*: crea un nuevo usuario con el nombre recibido como parámetro.
- *deleteUser*: elimina un usuario del sistema mediante el ID recibido como parámetro.

Las funciones de prefijo “post” refieren a la inserción de documentos, en las cuales se reciben los campos en el parámetro “req”, se crea la instancia del modelo y se invoca el método *save()*, devolviendo NULL en caso de error.

Las funciones de prefijo “delete” se encargan de borrar un documento determinado a partir del ID en el request, invocando la función *deleteOne()*.

Por ejemplo, la función *deleteCard* resulta de la siguiente manera:

```
deleteCard: async(req, res) => {  
    const id = req.params.id  
    if (!id) return res.status(400).json({message: "ID no especificado"})  
    const card = await model.findById(id)  
    if (!card) return res.status(400).json({  
        message: "ID no encontrado en la base de datos"})  
    const cuenta = await cuentaModel.findOne({tarjeta: id})  
    if (cuenta) await cuenta.deleteOne()  
    const result = await card.deleteOne()  
    return result ? res.json(result) : res.status(400).json({  
        message: "Error al borrar"})  
}
```

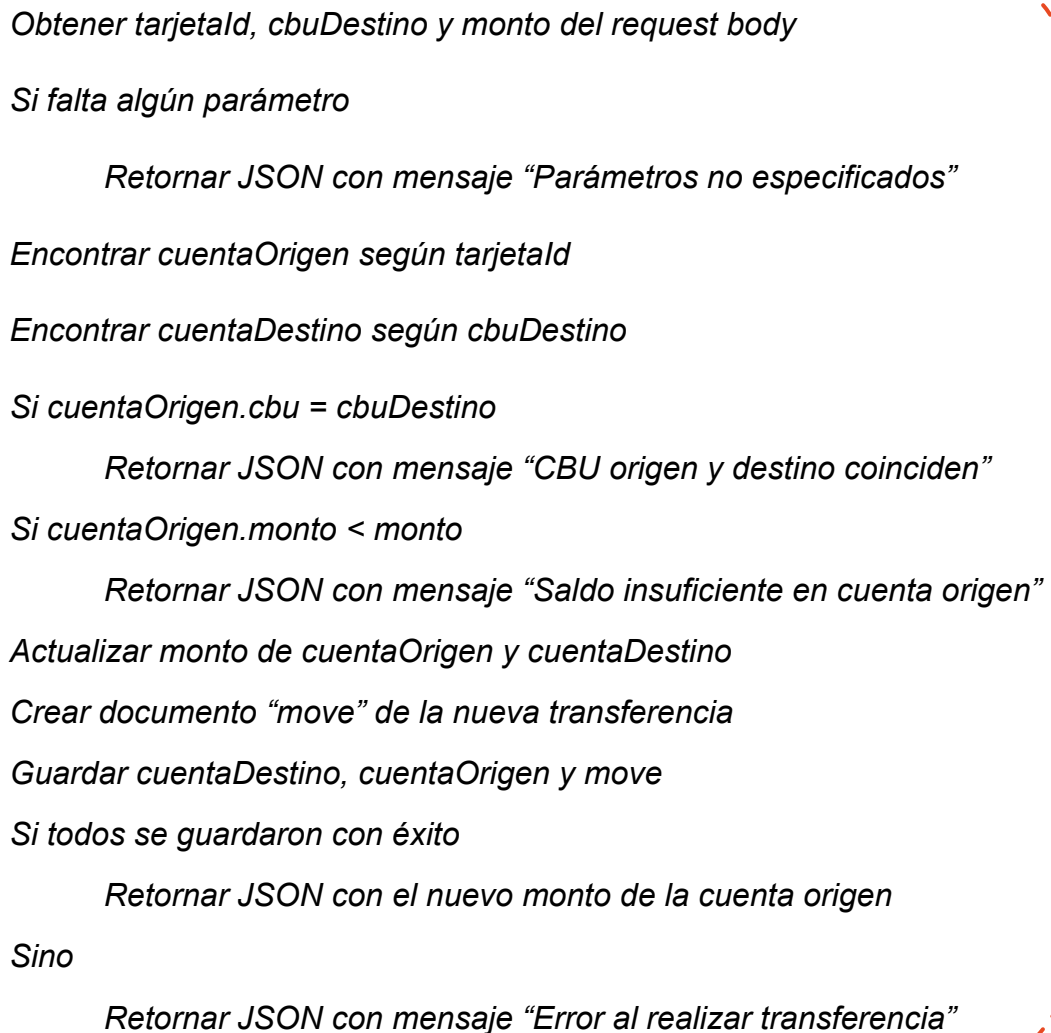
Código 6.4. Eliminación de tarjeta y su cuenta asociada (si existe)



Como se puede observar, cada cuenta siempre es creada y eliminada junto a su tarjeta asociada, al poseer una cardinalidad (1,1). En su creación predeterminada, se le asigna el tipo “Cuenta corriente en pesos” y monto \$0.

Otras funciones, como banearTarjeta y desbanearTarjeta, solo realizan modificaciones a uno o más campos del documento, encontrando el mismo por ID y explicitando los nuevos valores con la función findByIdAndUpdate(), y luego realizando la persistencia en la base de datos mediante el método save().

Por último, cabe destacar que la función “postMove” realiza una combinación de las operaciones de inserción y actualización entre documentos de diferentes colecciones, como se muestra a continuación.



```
Obtener tarjetaId, cbuDestino y monto del request body
Si falta algún parámetro
    Retornar JSON con mensaje "Parámetros no especificados"
Encontrar cuentaOrigen según tarjetaId
Encontrar cuentaDestino según cbuDestino
Si cuentaOrigen.cbu = cbuDestino
    Retornar JSON con mensaje "CBU origen y destino coinciden"
Si cuentaOrigen.monto < monto
    Retornar JSON con mensaje "Saldo insuficiente en cuenta origen"
Actualizar monto de cuentaOrigen y cuentaDestino
Crear documento "move" de la nueva transferencia
Guardar cuentaDestino, cuentaOrigen y move
Si todos se guardaron con éxito
    Retornar JSON con el nuevo monto de la cuenta origen
Sino
    Retornar JSON con mensaje "Error al realizar transferencia"
```

*Pseudocódigo 6.1. Función postMove de moveController*

### 6.1.3 – Rutas

Cada función definida en los controladores es accedida a partir de una URL asociada de la API utilizando los métodos de Router de Express. Esta correspondencia está indicada en los archivos de sufijo “Api” del directorio **routes**, como se muestra en las tablas a continuación. Los casos donde la ruta indica “/:” se trata de un request param que se debe pasar, como “/:id”.

- URL base para tarjetas: “localhost:2000/api/cards”.

Ruta	Tipo	Función	Response
/count	GET	getCount	count: Number
/all	GET	getAll	tarjetas: Array
/pin/:nro	GET	getPin	pin: Number
/addcard	POST	postCard	Tarjeta agregada con ban: false y un CBU autogenerado
/ban/:id	PATCH	banearTarjeta	Tarjeta actualizada
/unban/:id	PATCH	desbanearTarjeta	Tarjeta actualizada
/delete/:id	DELETE	deleteCard	Tarjeta borrada

- URL base para cuentas: “localhost:2000/api/cuentas”. En los casos donde la respuesta es un monto, se trata del monto actualizado de la cuenta luego de la operación si resultase exitosa.

Ruta	Tipo	Función	Response
/all	GET	getAll	list: Array
/count	GET	getCount	count: Number
/monto/:tarjeta	GET	getMonto	monto: Number
/ingreso	POST	ingresarMonto	monto: Number
/retiro	POST	retirarMonto	monto: Number
/cbu-info/:tarjetaId/:cbuTarget	GET	getCbuInfo	Nombre del cliente de cbuTarget

- URL base para transacciones: “localhost:2000/api/moves”.

Ruta	Tipo	Función	Response
/all	GET	getAll	movimientos: Array
/count	GET	getCount	count: Number
/transferir	POST	postMove	monto: Number

- URL base para clientes: “localhost:2000/api/users”

Ruta	Tipo	Función	Response
/all	GET	getAll	Usuarios: Array
/count	GET	getCount	count: Number
/adduser	POST	postUser	Cliente agregado
/delete/:id	DELETE	deleteUser	Cliente borrado

Para las peticiones tipo POST, se requieren especificar los siguientes parámetros en el “body” para que la operación se realice correctamente.

API	Ruta	Request Body	Notas
<b>cards</b>	/addcard	clienteSeleccionado: String nro: String pin: Number fechavto: Date cvv: Number	El primer parámetro es el ID del cliente titular
<b>cuentas</b>	/ingreso	tarjetaId: String monto: Number	ID de tarjeta en la base de datos
<b>cuentas</b>	/retiro	tarjetaId: String monto: Number	ID de tarjeta en la base de datos
<b>moves</b>	/transferir	tarjetaId: String cbuDestino: Number monto: Number	ID de tarjeta origen en base de datos
<b>users</b>	/adduser	nombre: String	-

### 6.1.4 – Configuración MQTT

En el `index.js` además se instancia el cliente MQTT de la PC en la función `mqttConfig()` para poder comunicarse con el programa del cajero implementado en la Raspberry Pi, utilizando la librería MQTT para NodeJS. Como el bróker se encuentra en el mismo equipo que el sistema back-office, al momento de invocar el método `connect()` se pasa la dirección IP de `localhost` (`127.0.0.1`) y el puerto **1883**, que es el predeterminado para este protocolo [9].

De acuerdo a la siguiente tabla de tópicos, si la conexión resulta exitosa, se procede a suscribirse a aquellos donde el publicador es la Raspberry, mediante el método `subscribe()` del cliente MQTT.

Nombre del tópico	Publicadores	Mensaje
<b>cajero/efectivo</b>	Ambos	Efectivo en cajero
<b>cajero/limites</b>	Ambos	Mínimo - Máximo de extracción
<b>cajero/status</b>	Raspberry	0 (desconectado) 1 (conectado)
<b>cajero/pin_request</b>	Raspberry	Nro. de tarjeta
<b>cajero/pin_response</b>	Backend	PIN de la tarjeta
<b>cajero/monto_request</b>	Raspberry	ID tarjeta
<b>cajero/monto_response</b>	Backend	Saldo en cuenta
<b>cajero/ingreso_request</b>	Raspberry	ID tarjeta + monto
<b>cajero/ingreso_response</b>	Backend	Nuevo saldo
<b>cajero/retiro_request</b>	Raspberry	ID tarjeta + monto
<b>cajero/retiro_response</b>	Backend	Nuevo saldo
<b>cajero/cbu_request</b>	Raspberry	ID tarjeta + CBU destino
<b>cajero/cbu_response</b>	Backend	Nombre propietario cuenta destino
<b>cajero/transfer_request</b>	Raspberry	ID tarjeta + CBU destino + Monto
<b>cajero/transfer_response</b>	Backend	Nuevo monto de cuenta origen

Luego, se indica la acción a realizar para cada mensaje recibido en el método `on("message")` del cliente, filtrando por nombre de tópico.

En los de sufijo “request”, se realiza un llamado a la API de la sección 6.1.3, publicándose la respuesta en el tópico de sufijo “response” tanto en caso de éxito (then) como de fallo (catch). Antes del llamado es necesario realizar una separación de los datos del mensaje recibido mediante `split()` si éste es compuesto, es decir, contiene al menos un guion como delimitador.

Tópico suscrito	Partes	Ruta API	Tópico respuesta
<b>pin_request</b>	1	cards/pin	pin_response
<b>monto_request</b>	1	cuentas/monto	monto_response
<b>ingreso_request</b>	2	cuentas/ingreso	ingreso_response
<b>retiro_request</b>	2	cuentas/retiro	retiro_response
<b>cbu_request</b>	2	cuentas/cbu-info	cbu_response
<b>transfer_request</b>	3	moves/transferir	transfer_response

Los otros tres tópicos suscritos corresponden a valores globales del cajero, no asociados a documentos de la base de datos de MongoDB. Se describe el procesamiento de cada uno a continuación.

- **cajero/efectivo:** se recibe el valor de efectivo actual en el cajero, tanto al iniciarse el cajero como luego de cada operación de ingreso y retiro. Este valor se guarda en una variable local de `index.js` llamada “efectivo”, y se emite por socket para visualizarlo en el frontend sin recargar la página.
- **cajero/limites:** se recibe un mensaje compuesto con los valores mínimo y máximo de los límites de extracción al iniciarse el programa del cajero, guardándose los mismos en un arreglo local “límites” del `index.js`.
- **cajero/status:** se recibe el dígito “1” cuando el programa del cajero en la Raspberry se encuentra en ejecución, o bien, el dígito “0” cuando se finaliza la ejecución del mismo. De igual manera que el efectivo, el valor se guarda en una variable local y se emite por socket al frontend.

### 6.1.5 – Configuración Socket

El socket mencionado en la sección anterior se configura en la función **webSocketConfig()** del index.js. La necesidad de implementarlo surge de que debe existir una forma de actualizar de manera automática el valor de efectivo y estado recibido por MQTT en la página web que ve el administrador. Por ello, se decidió importar la librería “socket.io” para dicho “reenvío” [10].

En consecuencia, también se debe importar “http” para la conexión del servidor y se debe especificar el puerto correspondiente al frontend (3000). Una vez inicializado, al conectarse un usuario al socket se guarda dicha instancia en la variable “miSocket” del index.js. El código resulta de la siguiente manera:

```
const { Server } = require("socket.io");  
  
// ...  
  
const server = require("http").createServer(app);  
  
const io = new Server(server, {  
  cors: {  
    origin: "http://localhost:3000",  
    methods: ["GET", "POST"]  
  }  
});  
  
io.on("connection", (socket) => miSocket = socket);  
  
// ...  
  
server.listen(2000, () => console.log("Servidor en ejecución"));
```

*Código 6.6. Configuración del socket en backend*

### 6.1.6 – API completa

Mediante el método `use()` de Express, se agregan todas las rutas disponibles desde el backend. De esta forma, primero se indicó un middleware que muestra por consola la URL para todo llamado realizado a la API.

Luego, se incorporaron los 4 archivos de la sección 6.1.3, explicitando la URL base de cada uno, por ejemplo: `"/api/users"` para `userApi`. También se agregaron nuevas rutas tipo GET para recuperar valores recibidos por MQTT:

- `"/api/status"`: retorna true o false según el estado del cajero.
- `"/api/cash"`: retorna el valor de la variable efectivo, por defecto 0.
- `"/api/settings/limites"`: retorna un JSON con los límites mínimo y máximo de extracción del cajero, en los campos `"min"` y `"max"`, respectivamente.

Se agregó también la solicitud tipo POST para el efectivo y los límites, con el fin de poder modificar estos valores desde la página de configuración del administrador. En ambos casos, se requiere que los campos (min y max para límites, value para efectivo) existan en el body del request, y luego se publican los nuevos valores por el tópico correspondiente de MQTT. La respuesta para el frontend es un JSON con un mensaje a mostrar según el éxito de la operación.

Finalmente, el API Rest del backend se resume de esta forma:

Ruta	Tipo	Response
<code>/api/cards/</code>	*	Dada por <code>cardController</code>
<code>/api/cuentas/</code>	*	Dada por <code>cuentaController</code>
<code>/api/moves/</code>	*	Dada por <code>moveController</code>
<code>/api/users/</code>	*	Dada por <code>userController</code>
<code>/api/status</code>	GET	value: Boolean
<code>/api/cash</code>	GET	value: Number
<code>/api/cash</code>	POST	value: Number
<code>/api/settings/limites</code>	GET	{min: Number, max: Number}
<code>/api/settings/limites</code>	POST	message: String

## 6.2 – Frontend

Constituye la interfaz web gráfica para el administrador, desarrollado con el framework ReactJS y ejecutado también con Node en el puerto **3000**. El nombre del programa principal también es **index.js** de la carpeta “src”, y para su ejecución se utiliza el comando “npm start” en el directorio frontend.

Para poder utilizar la API ya desarrollada de manera simple, se importó la librería Axios que proporciona directamente los métodos get, patch, post y delete, en lugar de utilizar fetch() de JavaScript. Además, para evitar repetir la dirección base “127.0.0.1:2000/api” y el timeout en cada llamado, se decidió exportar desde el index.js una instancia “miApi” de Axios con dicha configuración [11].

En este archivo, también se renderiza el componente App, explicado en la sección siguiente, como el elemento root de React.

### 6.2.1 – App

Es el componente principal, implementado en el archivo App.js del directorio src, donde se hace uso del hook useState() para guardar el estado del tema actual en “darkMode”. También se utiliza el hook useEffect() para modificar los valores de propiedades CSS cada vez que cambia el tema.

En todas las páginas está presente un header, representado con el componente **Banner**, cuyo código se encuentra en el archivo JS homónimo del subdirectorio banner. El mismo posee el logo y nombre de aplicación, que permite volver al inicio al presionarlo; y también un ícono para alternar entre un tema claro y oscuro. Mediante el uso de “props” (parámetros de componentes), se conoce el valor de darkMode y se puede modificar pasando una función switchModeFn, que se encuentra implementada en el componente App.



*Figura 6.2. Componente “Banner” y sus dos estados posibles.*



Cada página tiene asociada una URL particular, y desde la App se muestra el componente correspondiente a dicha página. Para ello, se utiliza la librería React Router DOM, estableciéndose un BrowserRouter como elemento raíz y luego los paths con sus páginas asociadas dentro de Routes. En el caso del Banner, al ser siempre visible, se extrae del procesamiento de rutas.

```

<BrowserRouter>

  <Banner darkMode={darkMode} switchModeFn={switchMode} />

  <Routes>

    <Route path="/" element={<HomePage/>} />

    ...

  </Routes>

</BrowserRouter>

```

*Código 6.7. Estructura del BrowserRouter en App*

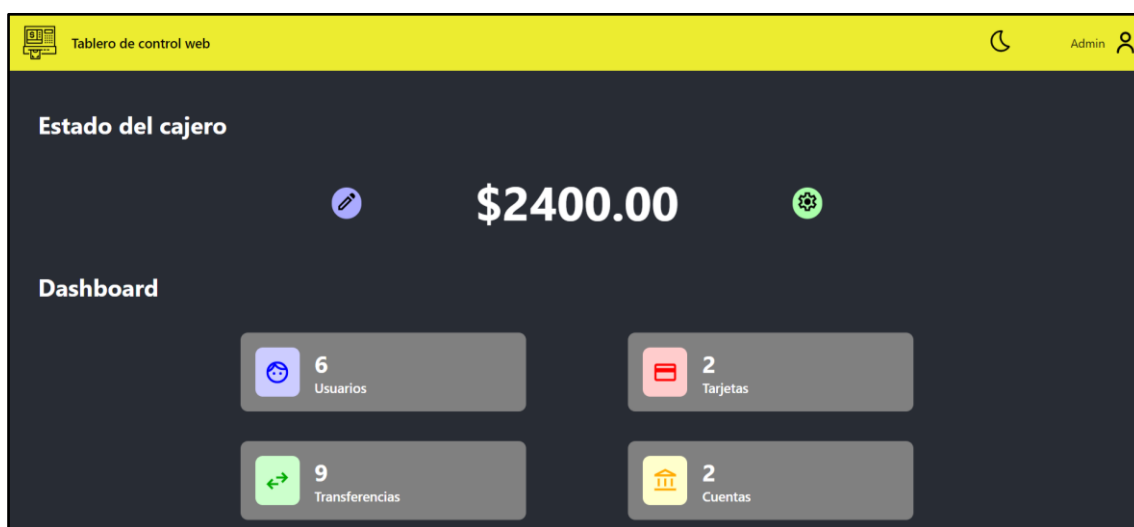
Las páginas incluidas dentro de la etiqueta <Routes> son las siguientes:

Path	Element	Descripción
/	HomePage	Pantalla principal donde se muestra el efectivo en cajero y un dashboard con las cantidades de cada colección
/users	UserListPage	Lista de clientes registrados
/cards	CardListPage	Lista de tarjetas registradas
/moves	MoveListPage	Lista de transferencias realizadas
/accounts	AccountListPage	Lista de cuentas registradas
/users/add	AddUserPage	Formulario para agregar cliente
/cards/add	AddCardPage	Formulario para agregar tarjeta
/settings	SettingsPage	Configuración de límites de extracción
*	Page404	Redirección por URL desconocida

Por otra parte, cuando se requiere acceder a una URL del frontend, por ejemplo, al presionar un botón, se utiliza un componente Link en lugar del anchor de HTML, ya que de esta manera se evita recargar la App en su completitud.

### 6.2.2 – HomePage

Corresponde a la pantalla principal del back-office, cuyo código se encuentra en el subdirectorio home. Consiste en un dashboard, donde se puede ver el estado actual del cajero, ingresar a un menú de opciones y acceder a las diferentes tablas de cada colección explicada en la sección 6.1.1.



*Figura 6.3. Página principal con estado activo del cajero.*

En su código, posee un total de 8 estados, de los cuales 4 se utilizan como contadores de cada colección, otros 2 aplican para el editor de efectivo (modal abierto y valor del cuadro de diálogo) [22], y los últimos 2 corresponden al “status” y efectivo disponible del cajero. Estos valores se establecen al cargar la página en el hook `useEffect`, realizando llamados a la API del backend mediante `Axios`, y también haciendo uso del método `socket.on()`, de la librería `socket.io-client`, para actualizar el status y efectivo de manera inmediata al recibirse por MQTT.

La conexión al socket desde la parte frontend resulta sencilla al realizarse mediante una única línea de código:

```
const socket = io.connect("http://localhost:2000");
```

Para mostrar cómo se complementan las peticiones GET de Axios con la recepción de datos vía socket se muestra parte del código de `useEffect`:

```
useEffect(() => {  
    miApi.get("users/count")  
        .then(res => setUserCount(res.data.count))  
        .catch(err => console.error(err));  
    // ...  
    miApi.get("cash")  
        .then(res => setCash(res.data.value))  
        .catch(err => console.error(err));  
    socket.on("cash", data => setCash(data.value));  
}, []);
```

*Código 6.8. Fragmento del hook `useEffect` en `HomePage`*

El valor de “status” se utiliza para mostrar el efectivo junto al acceso a la pantalla de opciones y el editor de efectivo (modal) si el cajero está activo, o bien, el texto “Desconectado” en su lugar en el caso contrario.

Respecto al dashboard, cada botón es un componente **DashboardCard** (código en el mismo directorio `home`) que contiene un ícono y color particular de la colección, junto al nombre y el número de documentos registrados. Estos valores se pasan a través de las props, así como también la URL de la página a cargar cuando se presiona en el mismo.

Los íconos están en formato SVG y se encuentran descargados en el directorio “src/assets”. Para su utilización, se importan en el archivo de React donde se utilizan, estableciendo un nombre particular y la ruta local del mismo.

### 6.2.3 – Listados

Cada colección tiene su directorio asociado en el frontend, siendo “cards” para tarjetas, “accounts” para cuentas, “moves” para transferencias y “users” para usuarios. Luego, en cada directorio existe un archivo con sufijo “ListPage”, que corresponde a la página que muestra una tabla de la colección completa, siguiendo una estructura similar, que se describe a continuación.

Se utiliza una variable de estado para almacenar todos los documentos en un arreglo, realizando su carga en el hook `useEffect`.

En la parte superior izquierda, siempre se muestra el componente **PageHeader** del directorio “src/common”, que consiste en el ícono y color mostrado en el dashboard junto al nombre de la colección, a modo de título.

En la parte superior derecha, en los casos de `UserListPage` y `CardListPage` se agrega el componente **AddButton**, también del directorio common, que permite acceder al formulario de inserción de nuevo documento.



Nombre y Apellido	Tarjetas	Opciones
Juan Perez	1	Borrar
Alejo Martinez	0	Borrar
Agustin Medina	0	Borrar
Santiago Sáenz	1	Borrar
Pablo Fernández	0	Borrar

*Figura 6.4. Lista de usuarios, componente `UserListPage`*

Como una colección puede llegar a contener una gran cantidad de documentos, se agregó también un estado “loading” para mostrar una animación de carga mientras no se haya completado la petición GET del `useEffect`. Dicho estado es una variable booleana inicializada en `TRUE`, y se utiliza con un operador ternario para mostrar la animación de carga o la tabla completa.

A efectos estéticos, si se disponen de pocos datos esta carga puede ser muy rápida, por lo que se optó una duración mínima de 200 ms de animación; a excepción de que no haya documentos en la colección consultada u ocurra un error, como la desconexión del backend, mostrando el mensaje “Sin datos”.

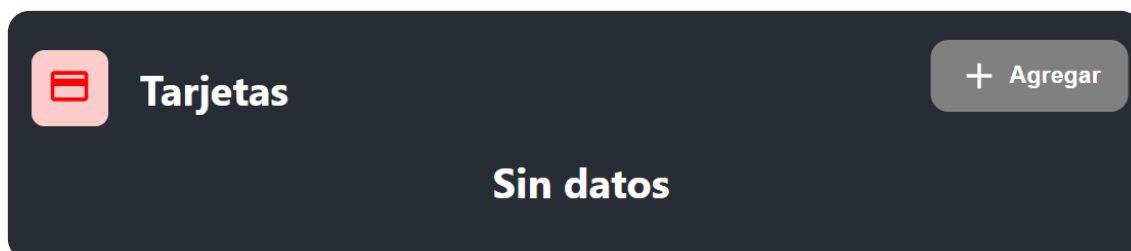


Figura 6.5. Lista de tarjetas en caso de error o tabla vacía.

#### 6.2.4 – Armado de tablas

Una vez obtenidos los documentos vía API se muestra cada uno en una fila de tabla. Para ello, se emplea la función map, retornando un elemento `<tr>` para cada ítem, y cada fila contiene los datos que se indican a continuación.

Listado	Columnas	Descripción
Usuarios	Nombre y apellido	Campo “nombre”
	Tarjetas	Número de tarjetas asociadas
	Opciones	Borrar
Tarjetas	Número	Campo “nro” con espacios
	Estado	Habilitado o Bloqueado
	Opciones	Banear/Activar y Borrar
Transferencias	Fecha	Campo “_id” como fecha y hora
	Origen	Nombre del cliente origen
	Destino	Nombre del cliente destino
	Monto	“\$” + campo “monto”
Cuentas	CBU	Campo “cbu”
	Alta	Campo “_id” como fecha
	Cliente	Nombre del cliente
	Saldo	“\$” + campo “monto”

Las “opciones” son un conjunto de botones que permiten realizar alguna acción sobre el ítem en particular, como su borrado de la base de datos, y son los componentes nombrados **DeleteButton**, **BlockButton** y **UnlockButton**. Una excepción al borrado sucede cuando se intenta eliminar un cliente que posee alguna tarjeta registrada a su nombre. Luego de presionar “Borrar”, se mostrará un mensaje informando que dicha acción no se puede realizar.

Fecha	Origen	Destino	Monto
Nov 20 2023 19:17	Alejo Martínez	Agustín Medina	\$1000.00
Nov 23 2023 18:58	Juan Perez	Santiago Sáenz	\$200.00

*Figura 6.6. Tabla de transferencias con datos de ejemplo.*

Para la columna de fecha que se observa en la Fig. 6.6, y que también existe en la tabla de Cuentas (columna “Alta”), se tuvo en cuenta que no es necesario almacenar el timestamp como un campo adicional en la base de datos de MongoDB. Esto se debe a que la fecha y hora completa del momento en que se guardó el documento se puede obtener desde el ID, que es un String de representación hexadecimal [14]. La conversión se realiza en la función `getDate`, pasando el ID como parámetro, cuyo código se muestra a continuación.

```
function getDate(id){  
    return new Date(parseInt(id.substring(0,8), 16) * 1000)  
        .toString().substring(3,21);  
}
```

*Código 6.9. Función de conversión de ID a fecha y hora*

### 6.2.5 – Formularios

Se crearon los formularios para agregar usuarios y tarjetas, en los archivos **AddUserPage** y **AddCardPage**. Para las tarjetas, se realiza además una petición GET para obtener el listado de nombres de clientes, y se muestra un componente **CreditCard** con apariencia realista que se va completando a medida que el administrador rellena los campos, mostrado en la Figura 6.7.



Formulario "Agregar tarjeta" en tema claro. El formulario contiene una visualización de una tarjeta de crédito simulada y campos de entrada para los datos de la tarjeta.

La tarjeta simulada muestra:

- Número de tarjeta: 4567 9812 3456 7890
- VALIDO HASTA: 06 / 26
- USUARIO EJEMPLO
- Logotipo de Mastercard

Los campos de entrada son:

- Número de Tarjeta:
- PIN:
- Fecha de Vencimiento:  (con icono de calendario)
- CVV:
- Cliente:  (con flecha de selección)

Botón: **Agregar Tarjeta**


Figura 6.7. Formulario "Agregar tarjeta" en tema claro.

Se controla la cantidad y tipo de caracteres, y también que se hayan completado los campos requeridos al presionar el botón de envío (función `handleSubmit`), realizando la petición POST del nuevo documento y cargando el listado correspondiente mediante la función `navigate()` de React.

### 6.2.6 – Configuración de límites

A partir del ícono de la “rueda” del HomePage, se accede a una página para actualizar los límites de extracción, llamada **SettingsPage** en “src/settings”.

En el useEffect, se cargan los valores actuales del cajero mediante una petición GET de Axios. Al presionar el botón “Actualizar” se verifica que ambos campos estén completos, y la función handleSubmit controla que los valores sean positivos y que el mínimo no supere al máximo. Si dichas condiciones se cumplen, se realiza un POST al backend, para que luego se informe al cajero vía publicación MQTT, tal como se mostró en la sección 6.1.4. Además, se muestra un mensaje según el éxito de la operación, y luego se retorna al HomePage.



The screenshot shows a dark-themed form titled "Configuración de límites de retiro". It contains two input fields: "Mínimo" with the value "1000" and "Máximo" with the value "50000". Below these fields is a blue button labeled "Actualizar".

*Figura 6.8. Apariencia normal de SettingsPage.*



The screenshot shows the same "Configuración de límites de retiro" form, but with the "Mínimo" field set to "10000" and the "Máximo" field set to "500". Below the "Máximo" field, a red error message reads: "El máximo no puede ser menor que el mínimo." The "Actualizar" button is still present.

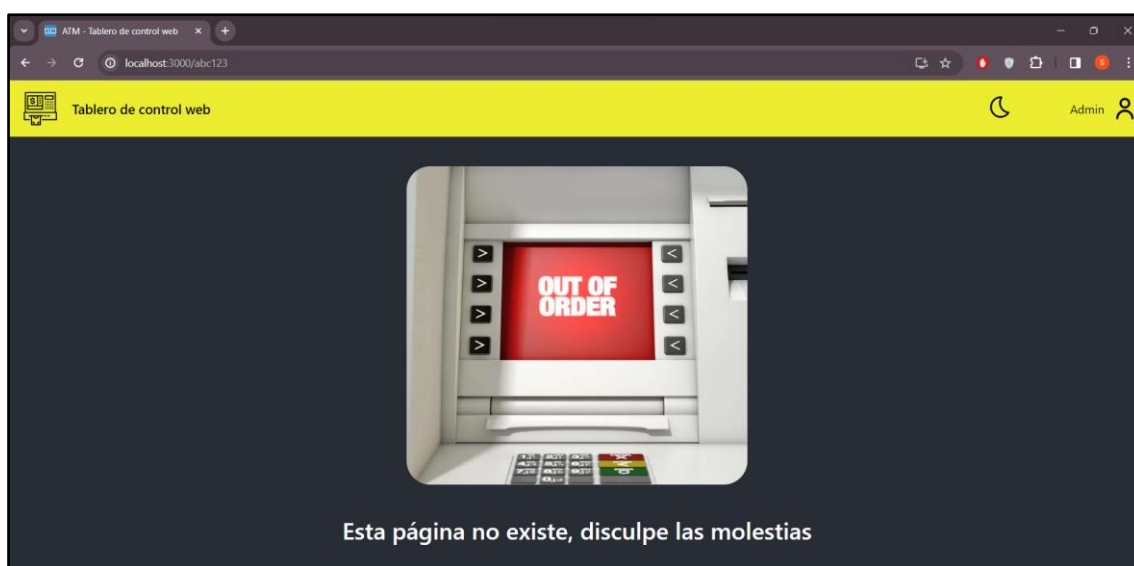
*Figura 6.9. Mensaje mostrado al intentar actualizar con valores invertidos.*



### 6.2.7 – Error 404

En caso de que el administrador manipule directamente la URL a partir de la barra de búsqueda del navegador, se cargará la página correspondiente a la misma según el BrowserRouter de App (sección 6.2.1). Si el “path” no se corresponde con ningún componente, se toma el caso “\*” donde la página a cargar es **Page404**, cuyo código se encuentra en el directorio “src/error”.

La página consiste en una imagen JPG ubicada en el mismo directorio y un texto con el mensaje “Esta página no existe, disculpe las molestias”.



*Figura 6.10. Página de error 404 mostrada para la URL “/abc123”*

### 6.2.8 – Otros archivos

Los archivos de extensión .css que aplican estilos a las páginas se encuentran en el mismo directorio que su correspondiente componente de React (archivo JSX). Sin embargo, los estilos globales, como fuente y colores según el tema actual, están definidos en index.css, mientras que algunas propiedades cuyos valores son comunes para varias páginas se encuentran en App.css, ya que el componente App es el elemento que contiene el BrowserRouter.

Para finalizar, en la carpeta “frontend/public” se encuentra el favicon a mostrar para toda la aplicación, junto al index.html con el título, el elemento root (div) y una sugerencia para habilitar JavaScript en caso de estar desactivado.

## 7 – Sistema cajero RPI

El script implementado para la lógica del cajero en la Raspberry Pi está realizado en Python, y se compone de una serie de clases para representar las distintas partes de la integración con el sistema back-office explicado. En las secciones siguientes se explican estas clases ordenado por dependencias.

### 7.1 – Archivos Python

#### 7.1.1 – Preferencias

El cajero utiliza ficheros de extensión “.txt” para almacenar algunos datos que requieran ser recuperados en una próxima sesión. En este proyecto, se trata de los límites de extracción, en un archivo “config\_limites.txt”, y del efectivo disponible en el cajero, en un archivo “config\_efectivo.txt”.

Se crearon entonces las dos respectivas clases, con una estructura similar: `LimitesConfig` y `CashPreference`. Ambas asignan sus atributos con valores por defecto en su constructor y poseen los siguientes 3 métodos:

- **cargar()**: en caso de que el archivo exista, se leen y asignan los valores en los atributos de la clase, caso contrario se crea el nuevo archivo.
- **guardar()**: sobrescribe el archivo con los valores actuales de atributos.
- **get\_for\_publish()**: devuelve los valores de atributos como tipo String, adecuado para la transmisión por MQTT, separándolos por un guion en el caso de los límites de extracción.

También se definieron los siguientes dos métodos para `CashPreference`:

- **sumar(diff)**: agrega el nuevo monto indicado al atributo efectivo.
- **restar(diff)**: sustrae el monto indicado al atributo mencionado.

#### 7.1.2 – Sesión

Se declara la clase homónima que posee los siguientes atributos:

- **id**: interpretado como número de tarjeta, es el ID leído por RFID.
- **text**: texto almacenado en la tarjeta leída por RFID.

- pin: corresponde al pin de la tarjeta, inicializado en -1.
- pin\_respondido: booleano que indica si hubo respuesta del backend.
- card\_database\_id: string que corresponde al ID de la tarjeta en MongoDB.
- error: mensaje tipo string que se puede mostrar en pantalla.

Se implementaron 2 métodos que se invocan cuando el backend brinda una respuesta al PIN Request. Los mismos son **cancel**, donde se asigna un mensaje de error indicado, y **set\_pin**, que guarda el pin y el card\_database\_id.

### 7.1.3 – Máquina de Estados Finito

Para administrar los diferentes escenarios del cajero ATM se diseñó una máquina de estados finito de Moore, con 9 estados definidos en el archivo **Estados.py**. El diagrama de la MEF se presenta a continuación.

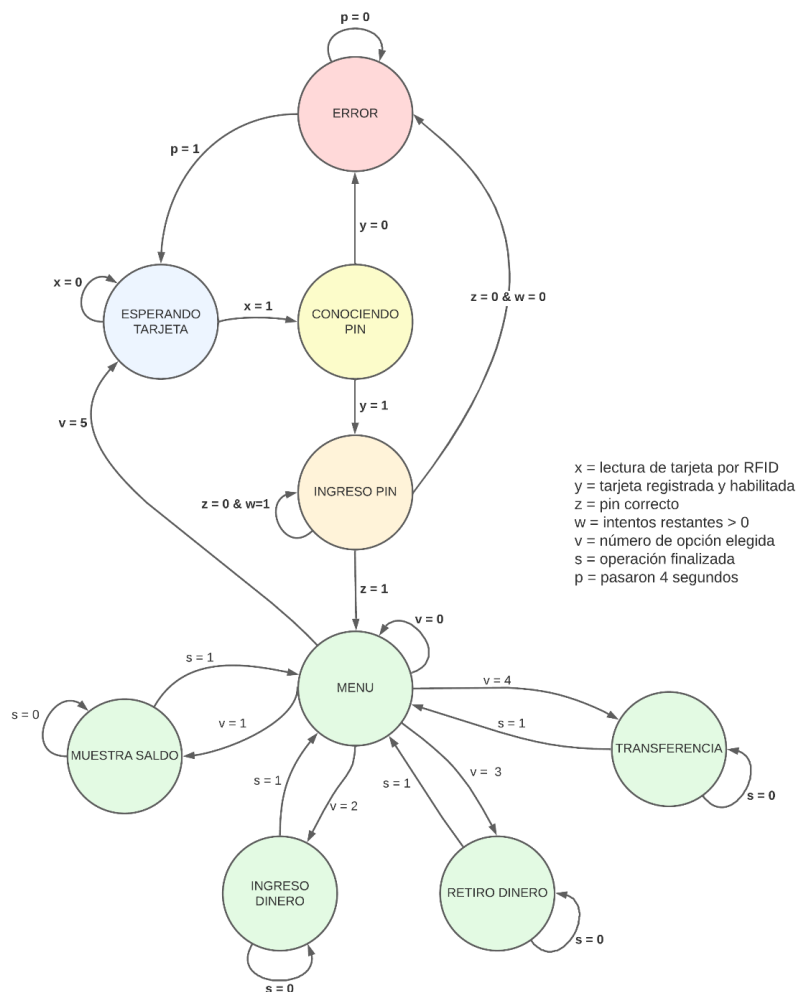


Figura 7.1. Máquina de estados finito del cajero automático.

La implementación se encuentra en el archivo **MEF.py**. En su constructor, se instancian las clases de preferencias (sección 7.1.1), se asigna el estado “Esperando tarjeta” como inicial, y se inicializan variables auxiliares como el contador de veces en estado, cantidad de intentos, monto, cbu y mensaje.

Se definieron los siguientes métodos:

- **start:** asigna el lector RFID, cliente MQTT, carga los límites de extracción y efectivo vía preferencias, y realiza 3 publicaciones MQTT para pasar dichos valores al backend de la PC e informar que el cajero está activo.
- **stop:** publica por MQTT que el cajero se encuentra inactivo e invoca los métodos guardar() de ambas clases de preferencias.
- **changeToState:** recibe el nuevo estado por parámetro, y lo asigna como actual, reiniciando los valores de las variables auxiliares.
- **getCurrentView:** retorna la URL de la página asociada al estado actual.
- **isCurrentView:** recibe una URL y retorna true si corresponde al estado actual, para evitar manipulación vía navegador.
- **update:** recibe opcionalmente un valor numérico de entrada “x” por parámetro (si no se indica vale -1) y realiza la acción correspondiente para la misma según el estado actual, como se muestra en la Fig. 7.1

A continuación, se describe cada estado:

- ❖ **Esperando tarjeta:** se mantiene en el mismo hasta la lectura efectiva de la tarjeta mediante el sensor RFID-Rc522.
- ❖ **Conociendo PIN:** a partir del número de tarjeta, se consulta al backend por MQTT si la misma está registrada y habilitada. En caso afirmativo se realiza la transición al estado “Ingreso PIN”, sino se muestra el mensaje de retroalimentación en el estado “Error”.
- ❖ **Ingreso PIN:** se solicita al usuario el ingreso por teclado del PIN correspondiente a la tarjeta hasta un máximo de 3 intentos. En caso de éxito se pasa al estado “Menú”, caso contrario al estado “Error”.

- ❖ **Menú:** se presentan las opciones disponibles del cajero (consultar saldo, ingresar dinero, retirar dinero, realizar transferencia y terminar), aguardando por una selección para pasar al estado asociado.
- ❖ **Muestra saldo:** se muestra el monto de la cuenta asociada a la tarjeta autenticada durante unos segundos, para luego volver al menú.
- ❖ **Ingreso dinero:** se muestra en pantalla un mensaje que le solicita al usuario el monto a ingresar, lo cual se indica por teclado. Una vez ingresado el valor, se presiona la tecla “Enter” y luego se publica por MQTT. Una vez recibido, se actualiza el nuevo saldo de la cuenta y se suma al efectivo del cajero.
- ❖ **Retiro dinero:** se muestra en pantalla un mensaje que le solicita al usuario el monto a retirar, de forma similar al depósito. Cuando el valor es ingresado y se presiona la tecla “Enter”, se envía por MQTT. Una vez recibido se actualiza el nuevo saldo de la cuenta y se resta del efectivo disponible del cajero.
- ❖ **Transferencia:** en primer lugar, se muestra un mensaje solicitando el ingreso del CBU destino. Al presionar “Enter”, en caso de que sea válido y distinto al origen, se solicita el monto a transferir, y luego al presionar “Enter” nuevamente se realizará la operación. En caso de error, se muestra un mensaje de retroalimentación.
- ❖ **Error:** consiste en un mensaje cargado desde código por una acción anterior, que se muestra en pantalla con un fondo rojo con duración de 3 segundos, para finalmente regresar al estado “Esperando tarjeta”.

#### 7.1.4 – Comunicación

El archivo declara una clase `Suscriptor`, cuya inicialización requiere pasar la instancia de la MEF de la sección anterior. Posee un método `suscribir_topicos()` donde se realiza la suscripción a los tópicos necesarios, cuyos nombres se encuentran en el archivo **Constantes.py**, donde también están aquellos utilizados para las publicaciones MQTT.

La suscripción a los tópicos se realiza mediante el método `subscribe()` de la clase `Client` de Paho-MQTT, indicando el nombre del tópico [15].

Por otra parte, el procesamiento de los mensajes se realiza en el método `procesar()`, que es invocado para cada mensaje recibido desde la función de callback “`onReceiveMqttMessage`” del programa principal (ver sección 7.1.5).

La acción realizada para cada tópico se describe a continuación.

Tópico suscrito	Procesamiento del mensaje
<b>limites</b>	Se asignan en los atributos de <code>LimitesConfig</code>
<b>pin_response</b>	En la instancia de Sesión, se guarda PIN y ID de tarjeta, o bien, se asigna como mensaje de error
<b>monto_response</b>	Se convierte a entero y se asigna a la variable “ <code>montoCuenta</code> ” de MEF
<b>ingreso_response</b>	
<b>retiro_response</b>	
<b>cbu_response</b>	Se asigna a variable “ <code>message</code> ” de MEF
<b>transfer_response</b>	Se convierte a entero y se asigna a “ <code>montoCuenta</code> ”
<b>cash</b>	Se asigna al atributo efectivo de <code>CashPreference</code>

### 7.1.5 – Programa principal

El script de la aplicación se denomina **app.py**, que requiere de un parámetro obligatorio para su ejecución desde consola. Se trata de la dirección IP de la computadora donde se encuentra el bróker MQTT. Se determinó que se realizara de esta forma, ya que dicha IP es distinta en cada red de Wi-Fi.

En el código, se crea una instancia de Flask (servidor en el puerto **5000** para la interfaz del cajero), como también de la MEF y la clase `Suscriptor`.

También se define una función `handle_signal()` para manejar la señal de interrupción “SIGINT” al presionar CTRL+C, donde se invoca el método `stop()` de la MEF, el método `cleanup()` de GPIO y se finaliza el programa con `exit(0)`.

Luego se tiene una serie de funciones que representan las vistas HTML a renderizar según la URL de Flask, y también el procesamiento a realizar para un conjunto de API Request, como la selección de una opción del menú.

Las vistas definidas son las siguientes:

Nombre de función	Ruta	HTML
<b>home</b>	/	Indicado por la función getCurrentView() de MEF
<b>error</b>	/error	error.html
<b>waiting_card</b>	/waiting-card	waiting_card.html
<b>pin_ack</b>	/pin-ack	waiting_card.html
<b>pin_input</b>	/pin-input	pin_input.html
<b>menú</b>	/menú	menu.html
<b>opcion_consulta</b>	/option-saldo	op_consulta.html
<b>opcion_ingreso</b>	/option-ingreso	op_ingreso.html
<b>opcion_retiro</b>	/option-retiro	op_retiro.html
<b>opcion_transferencia</b>	/option-move	op_move.html
<b>forward</b>	/forward	Redirección a "/"

Por otra parte, las funciones API Request implementadas son:

- **status:** tipo GET, devuelve un JSON con un único campo "status", que devuelve un string 'waiting' si el estado actual de la MEF es "Esperando tarjeta", en caso contrario devuelve 'ready'.
- **get\_datetime:** tipo GET, devuelve un JSON con un campo "text" que contiene la fecha y hora actual en formato string.
- **pin\_process:** tipo POST, requiere un parámetro "pin" con el valor ingresado en el cajero, actualiza la MEF con entrada x = 1 si el PIN es correcto o 0 en caso contrario. Al finalizar, devuelve un JSON con un campo "result", que contiene la ruta de la próxima vista a mostrar.
- **menu\_select\_option:** tipo POST, requiere un parámetro "option\_number" con el número de opción seleccionada, actualiza la MEF pasando como entrada dicho número, y al finalizar devuelve la ruta de la vista a mostrar.

- **api\_diff:** tipo POST, combina las operaciones de ingreso y retiro, recibe un parámetro “monto” que es asignado a la MEF, la actualiza con entrada  $x = 2$ , y al finalizar devuelve un JSON con un campo booleano “success” y otro llamado “msg” con el mensaje a mostrar en pantalla.
- **api\_consultar\_cbu:** tipo POST para la realización de transferencias, se recibe un parámetro “cbu” para asignárselo a la MEF y la actualiza la misma con entrada  $x = 2$ , esperando a que se complete la búsqueda del nombre de cliente asociado, y al finalizar se retorna un JSON con un campo booleano “success” y un mensaje en el campo “msg”.
- **api\_transferir:** tipo POST como continuación de la operación anterior, se recibe un parámetro “monto” que se asigna a la MEF, y la actualiza con entrada  $x = 3$ , esperando que se complete la transferencia para finalmente retornar un JSON con campos “success” y “msg”.
- **api\_get\_monto:** tipo GET para la consulta de saldo, que únicamente actualiza la MEF con entrada  $x = 2$  para que se realice la operación y luego devuelve un JSON con los campos “success” y “msg”.

Otra función definida es **onReceiveMqttMessage**, que únicamente pasa el tópico y payload del mensaje MQTT al método procesar() de Suscriptor.

Por último, se tiene una función **backgroundTasks** para ejecutar tareas en un segundo thread, dado que el método run() de la aplicación Flask es bloqueante. Posee una parte de setup, donde se instancia el lector RFID y el cliente MQTT indicando para éste último su ID, IP de bróker, tópicos a suscribir y función de callback, y se da inicio a la MEF pasando dichas instancias mediante el método start(); y luego posee un loop, que espera por la detección y lectura de una tarjeta de manera bloqueante siempre que el estado actual sea “Esperando tarjeta”, sino se ignora una actualización al pasar un valor de entrada  $x = -1$ , que resulta inválido.



Se tuvo en cuenta el empleo de una identificación aleatoria para el cajero, ya que se contempla el caso de que pudiera haber más de una Raspberry Pi ejecutando el mismo programa de Python.

```
lectorRfid = SimpleMFRC522()

cliente = mqtt.Client(f'cajero-{random.randint(0, 100)}')

cliente.connect(HOSTNAME)

suscriptor.suscribir_topicos(cliente)

cliente.on_message = onReceiveMqttMessage

cliente.loop_start()

mef.start(lectorRfid, cliente)

while 1:

    sleep(1)

    mef.update()
```

*Código 7.1. Función backgroundTasks() de app.py*

## 7.2 – Otros archivos

Los archivos HTML correspondientes a las diferentes vistas del cajero se encuentran en el directorio “templates”, mientras que los estilos CSS y códigos JavaScript de cada uno están organizados en carpetas según el tipo de archivo dentro del directorio “static”.

Se dispone también de archivos en “static/audio” que representan los sonidos producidos al presionar las teclas del cajero (keydown.mp3) y un leve ruido de fondo para el estado “Error” (error2-trim.mp3).

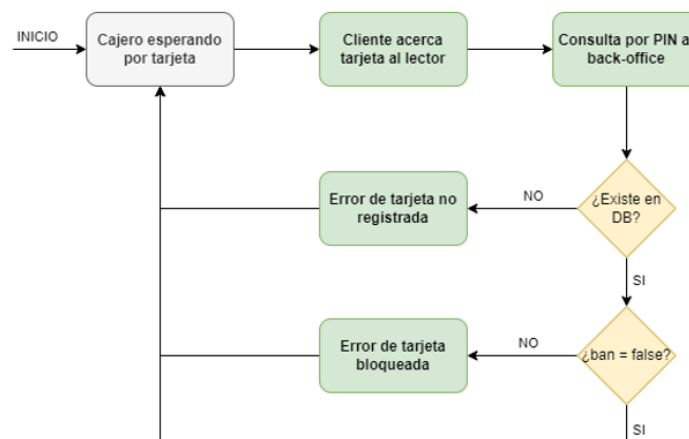
## 8 – Explicación de procesos

### 8.1 – Autenticación de cliente

Durante el estado “Esperando tarjeta” del cajero, el thread de background se encuentra en la ejecución bloqueante del método read() de SimpleMFRC522, hasta que efectivamente se acerque una tarjeta a dicho lector.

Por otra parte, el servidor de Flask permite el acceso a la página HTML correspondiente al estado mencionado. Al ser cargado en un navegador, se ejecuta también el script de JavaScript asociado a dicha página, que en este caso es “static/js/waiting\_card.js”, que consiste en un bucle de peticiones GET a la ruta “/status” de Flask cada 1500 ms. De esta forma, cuando el estado de la MEF sea distinto a “Esperando tarjeta”, la petición devuelve el String “ready”, redireccionando a “/” para mostrar la vista del nuevo estado.

Cuando finaliza la invocación de read(), en la MEF se consulta por el PIN al back-office, mediante una publicación MQTT en el tópico “cajero/pin\_request”, pasando como mensaje el ID de la tarjeta leída, interpretada como número de la misma, y se cambia al estado “Conociendo PIN”.



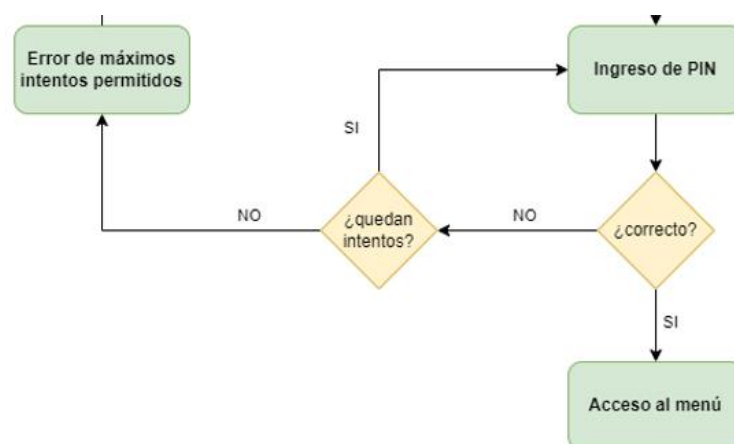
En el back-office, el mensaje MQTT es procesado mediante la invocación de la ruta “api/cards/pin” (método getPin de cardController), donde se comprueba primero que la tarjeta con el número indicado existe en la base de datos. En caso negativo, el controlador devuelve un JSON de estado 400 con el mensaje “Tarjeta no registrada en el sistema”.

La siguiente verificación en el controlador es el valor del campo “ban” de la tarjeta encontrada. Si es true, retorna un JSON de estado 400 con el mensaje “La tarjeta se encuentra bloqueada”; caso contrario, si está habilitada se retorna un JSON de 2 campos: “pin” y “tarjetaId”, correspondientes a la clave PIN y el ID de la tarjeta en la base de datos. Este último se añade para simplificar las consultas en las operaciones posteriores.

Si el controlador retorna estado 400, se publica desde el backend un mensaje de MQTT en el tópic “cajero/pin\_response” con el mensaje del JSON antecedido por un guion. En el caso de éxito, se publica por MQTT un mensaje compuesto de los campos “pin” y “tarjetaId” con un guion entre ellos, en el mismo tópic de respuesta.

Al recibirse el PIN en la Raspberry, se asigna la variable “pin\_respondido” de Sesión en true, y la variable “pin” resulta -1 si el mensaje comienza con un guion (caso error), realizando el cambio en la MEF al estado “Error” y mostrando el mensaje recibido en el navegador, caso contrario se pasa a “Ingreso PIN”.

La consulta de la variable “pin\_respondido” en cada iteración del estado “Conociendo PIN” se realiza un máximo de 5 veces (segundos), contemplando el caso de que no exista una respuesta del backend, pasando al estado “Error” con el mensaje “No hubo respuesta por parte del servidor”.



En el estado de “Ingreso PIN”, se muestra la vista HTML “pin\_input”, en cuyo código JavaScript se declara la función `addEventListener(“keydown”)` para almacenar los dígitos ingresados por teclado en el navegador, borrar el último si

se presiona la tecla Backspace, o realizar la petición POST a la ruta “/pin-process” si se presiona Enter y hay 4 dígitos ya ingresados.

La validación del PIN ingresado por el usuario con respecto al PIN de la tarjeta se realiza exclusivamente en la función de la ruta mencionada, actualizando la MEF con entrada  $x = 1$  si es correcto, provocando el cambio al estado “Menú”, o  $x = 0$  en caso contrario, incrementando el contador “attempts” de la MEF en una unidad, hasta un máximo de 3 veces, situación donde se realiza el cambio al estado “Error” mostrando el mensaje “Se alcanzó la máxima cantidad de intentos permitidos”.

## 8.2 – Consulta de saldo

Al seleccionar esta opción, en la MEF se envía el ID de la tarjeta a través de la publicación MQTT con el tópico “cajero/monto\_request”.

El backend es el responsable de gestionar la solicitud, consultando la base de datos para obtener el saldo correspondiente a la cuenta. En caso de que haya algún error como, por ejemplo, la inexistencia de la tarjeta o de la cuenta asociada de forma inesperada, se envía un valor de “-2” al tópico “cajero/monto\_response”, y se imprime un mensaje “No se pudo completar la operación” en la Raspberry. Por otra parte, en el caso de una operación exitosa, el backend publica el saldo en el mismo tópico, lo que provoca la aparición de un mensaje en la Raspberry que muestra el saldo actual de la cuenta.

La vista HTML del estado es “op\_consulta” y en su código de JavaScript se realiza el llamado a la ruta “api/monto” de Flask para iniciar la consulta de saldo en la MEF, y una redirección a “/forward” con un delay de 3 segundos.

## 8.3 – Ingreso de dinero

Al entrar a esta opción, en primer lugar, se le pide al usuario ingresar el monto a retirar para luego, realizar la comprobación de cantidad (si es número mayor a cero) y en caso afirmativo, se simula la operación y se realiza la publicación MQTT en el tópico “cajero/ingreso\_request”, pasando como mensaje el ID de la tarjeta seguido de un guion (-) y luego el monto ingresado.

El backend procesa la solicitud, realizando una petición POST a la ruta “api/cuentas/ingreso” con las partes del mensaje compuesto. El controlador busca la cuenta asociada en la base de datos y actualiza su monto. En caso de éxito se publica el nuevo saldo en el tópico “cajero/ingreso\_response”, sino se publica el mensaje “-2” como estado de error en el mismo tópico.

La Raspberry permanece en un bucle en espera de la confirmación. Si se recibió “-2”, se imprime “No se pudo completar la operación. Devolviendo el dinero ingresado”; sino se imprime “Operación realizada con éxito” seguido del nuevo saldo, y se publica el nuevo valor de efectivo en “cajero/efectivo”.

## **8.4 – Retiro de dinero**

Al entrar a esta opción, primero se le solicita al usuario que especifique por teclado el monto que desea depositar. Si este valor es cero, se considera que el usuario desea cancelar la operación, retornando al estado “Menú”.

Si se especifica un monto positivo, en la MEF se comprueba que el mismo no exceda al efectivo disponible en el cajero y que a su vez está dentro del rango permitido de extracción según los atributos de LimitesConfig. En caso de cumplir estas condiciones, se simula la operación y se realiza la publicación MQTT en el tópico “cajero/retiro\_request”, pasando como mensaje el ID de la tarjeta seguido de un guión (-) y luego el monto a retirar.

El backend procesa el mensaje, separándolo en sus partes por el delimitador guion e invocando un POST a la ruta “api/cuentas/retiro”. El controlador busca la cuenta a partir del ID de tarjeta y resta el monto únicamente si no excede el valor del saldo actual (es decir, no puede resultar negativo). En caso de éxito, se publica el nuevo saldo en el tópico “cajero/retiro\_response”, sino se publica el mensaje “-2” como estado de error en dicho tópico.

La Raspberry permanece en un bucle en espera de la confirmación. Si se recibió “-2”, se imprime “Extracción mayor al saldo disponible. Operación no realizada”; sino se imprime “Operación realizada con éxito” seguido del nuevo saldo disponible, y se publica el nuevo valor de efectivo en “cajero/efectivo”.

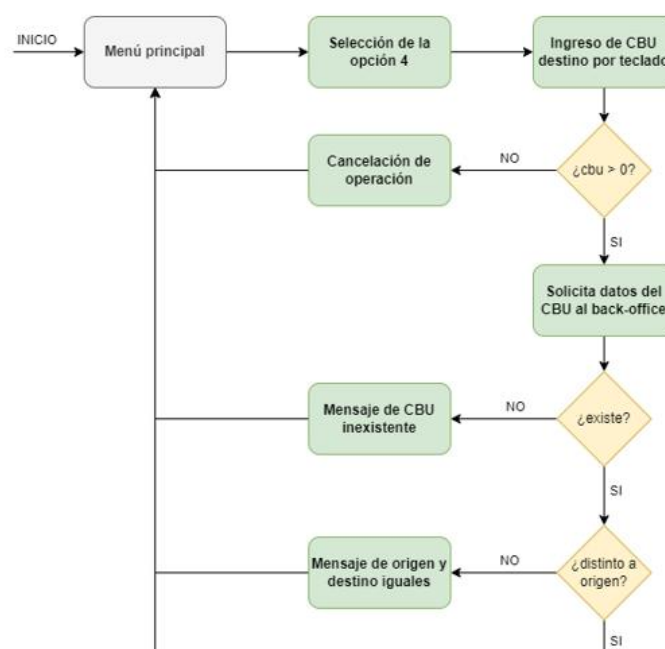
## 8.5 – Transferencia

Desde la vista de Menú, el cliente puede acceder a la opción de “Realizar transferencia” presionando la tecla 4 o bien mediante un clic en el botón en pantalla. De esta forma, se realiza la transición al estado “Transferencia” del cajero, cargando el HTML “op\_move” junto a su código JavaScript.

De igual manera que en las operaciones de ingreso y retiro, los dígitos se ingresan por teclado y se almacenan localmente en variables, pudiéndose borrar con la tecla Backspace y enviar a la API de Flask al presionar Enter.

El primer dato solicitado al cliente es el CBU de la cuenta destino, pudiendo especificar el valor 0 para cancelar la operación y volver al estado de Menú. La petición POST se realiza a la ruta “api/consultar-cbu”, pasando el número de CBU ingresado, que se guarda en la variable “cbu” de la MEF y se actualiza la misma con entrada x = 2. De esta manera, se publica un mensaje MQTT en el tópico “cajero/cbu\_request” compuesto de 2 partes: ID de tarjeta y CBU, que es recibido por el backend para invocar la ruta “api/cuentas/cbu-info”.

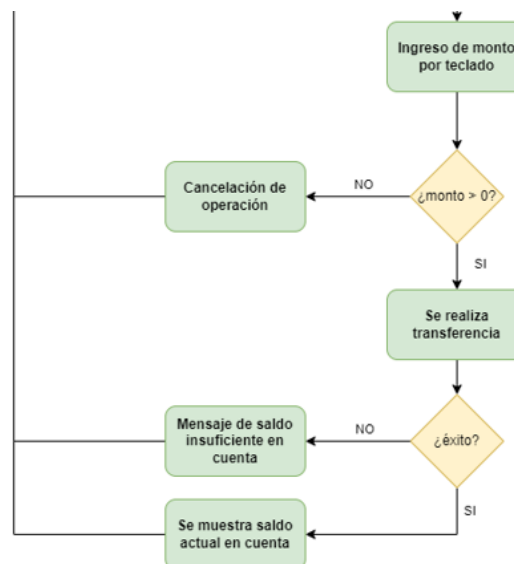
En la función getCbuInfo de cuentaController, se busca la cuenta destino a partir del CBU indicado y se verifica que la tarjeta asociada a dicha cuenta no sea la misma que la tarjeta utilizada para autenticarse en el cajero, ya que esto significa que las cuentas origen y destino corresponden al mismo cliente.



Si la búsqueda resulta exitosa, se retorna un JSON con el nombre y apellido del cliente titular de la cuenta destino (se realiza un populate de MongoDB para cruzar los campos de ambas colecciones). En MQTT, se publica en el tópic “cbu/response” dicho nombre, o bien, el mensaje de error antecedido por un guion para diferenciar los casos de éxito y fallo en la Raspberry.

La MEF espera de forma bloqueante la recepción del nombre, de modo que la petición POST realizada desde JavaScript (op\_move.js) retorna el mensaje indicado para mostrarlo directamente en el navegador.

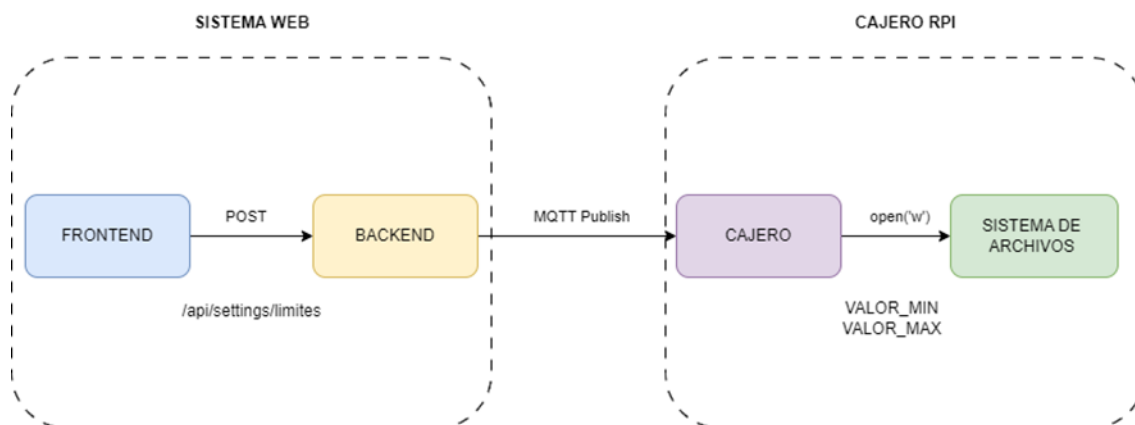
El siguiente dato a ingresar por el cliente es el monto a transferir, pudiendo cancelar la operación si se indica el valor cero. Al presionar Enter con un monto positivo, la realiza un segundo POST, pero a la ruta “api/transferir” de Flask, asignándose el valor ingresado en la variable “montoDiff” de la MEF y actualizando la misma con entrada x = 3 para diferenciarlo del caso CBU. La nueva publicación MQTT se realiza en el tópic “cajero/transfer\_request”, con un mensaje compuesto ahora además del monto a transferir, además del ID de tarjeta y el CBU destino (que sigue almacenado en la variable cbu).



El backend recibe el mensaje y lo procesa la función postMove del controlador de transferencias, presentada en el pseudocódigo 6.1. Se publica el nuevo saldo de la cuenta origen o el mensaje “-2” si ocurre un error, en el tópic “cajero/transfer\_response”, para finalmente ser mostrado en el navegador (el mensaje de error es un String constante en el código de op\_move.js).

## 8.6 – Actualización de límites

El siguiente esquema resume la secuencia de pasos para actualizar y persistir los límites de extracción entre el sistema web y el cajero en Raspberry.



*Figura 8.1. Secuencia de actualización de límites.*

La actualización de los límites se realiza desde el sistema web accediendo a la página de Settings desde el HomePage, solicitando al administrador ambos valores extremos de retiro de extracción. Una vez presionado el botón de actualizar, se realiza el POST de dichos valores donde en el backend se ejecuta una publicación MQTT en el tópico “cajero/limites” adjuntando el valor mínimo y el valor máximo, siempre y cuando el cliente MQTT se encuentre inicializado y se hayan especificado dichos montos en el Request Body.

En la función de callback de la clase Suscriptor (Raspberry) se reciben los valores para asignarlos en los atributos de LimitesConfig y persistirlos en el archivo “config\_limites.txt” mediante el método guardar(), separados por un salto de línea. De esta manera, en cada nuevo arranque del cajero se pueden recuperar los límites y se publican al backend, de forma que puedan ser cargados también al abrir la página de Opciones.



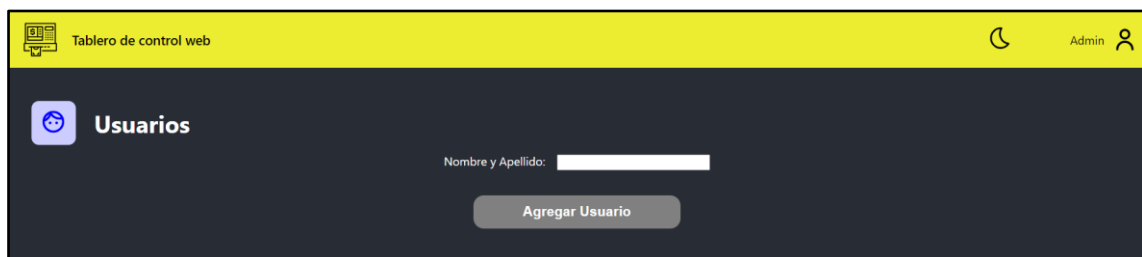
## 9 – Documentación relacionada

En el presente apartado, se muestran capturas de pantalla de las interfaces web desarrolladas tanto para el cajero ATM como el back-office, y se adjuntan vídeos de los procesos mostrados a lo largo del informe.

En la primera ejecución del sistema back-office, con el cajero desconectado y sin documentos en la base de datos, la página principal tendrá la apariencia de la Fig. 9.1. La misma situación ocurre si el servidor backend de la PC se encuentra desconectado, mostrando valores por defecto.



*Figura 9.1. Apariencia inicial de HomePage*



*Figura 9.2. Formulario de nuevo cliente, componente AddUserPage*

- ❖ **Borrado de clientes:** en caso de intentar borrar clientes con tarjetas asociadas, se mostrará una alerta al usuario sobre el problema, y si no posee será exitoso. Se puede observar este comportamiento en el video: <https://drive.google.com/file/d/1bxd1DaP9dchiVSCZGz2UCrOIDE37ug7J>

- ❖ **Registro de tarjeta:** en el formulario para agregar tarjeta, se verificó que todos los campos sean requeridos y tengan el formato correcto. Luego, se comprobó que la tarjeta aparezca en CardListPage, con las opciones disponibles, y también se haya creado la cuenta correspondiente, visible en AccountListPage. Por último, se intenta crear una tarjeta con el mismo número, lo cual termina siendo impedido por el sistema. Video: <https://drive.google.com/file/d/1W485UcSaFa7k458arluowLrwMDnw4Pki>

The screenshot shows a web control panel titled 'Tablero de control web' with a yellow header. On the left, there's a sidebar with a 'Tarjetas' icon. The main area displays a form for adding a new card. At the top of the form is a visual representation of a credit card with a chip, a magnetic stripe, and a Mastercard logo. Below this, the form includes input fields for 'Número de Tarjeta:', 'PIN:', 'Fecha de Vencimiento:' (with a date picker), 'CVV:', and a dropdown for 'Cliente:' labeled 'Selecciona un cliente...'. A grey button labeled 'Agregar Tarjeta' is at the bottom.

Figura 9.3. Formulario de nueva tarjeta, componente AddCardPage

The screenshot shows the 'Tarjetas' section of the web control panel. It features a table with three columns: 'Número', 'Estado', and 'Opciones'. There are three rows of card data. Above the table, there is a '+ Agregar' button. The table data is as follows:

Número	Estado	Opciones
4214 1241 3531 5135	Habilitado	<button>Banear</button> <button>Borrar</button>
5351 3515 2453 4534	Bloqueado	<button>Activar</button> <button>Borrar</button>
2141 4315 1351 3531	Bloqueado	<button>Activar</button> <button>Borrar</button>

Figura 9.4. Tabla de tarjetas, componente CardListPage

- ❖ **Primer arranque del cajero:** para iniciar el programa app.py se obliga a indicar la dirección IP del bróker MQTT por parámetro. En la primera ejecución en la Raspberry Pi, se crean los ficheros TXT con valores por defecto para el efectivo (\$2000) y límites de extracción (mínimo \$1000 y máximo \$5000), que se observan también en el panel back-office. Video: [https://drive.google.com/file/d/1ZkeBFpSBuEbpm7C\\_qPLMDg-XXjXWQCcD/view?usp=sharing](https://drive.google.com/file/d/1ZkeBFpSBuEbpm7C_qPLMDg-XXjXWQCcD/view?usp=sharing)

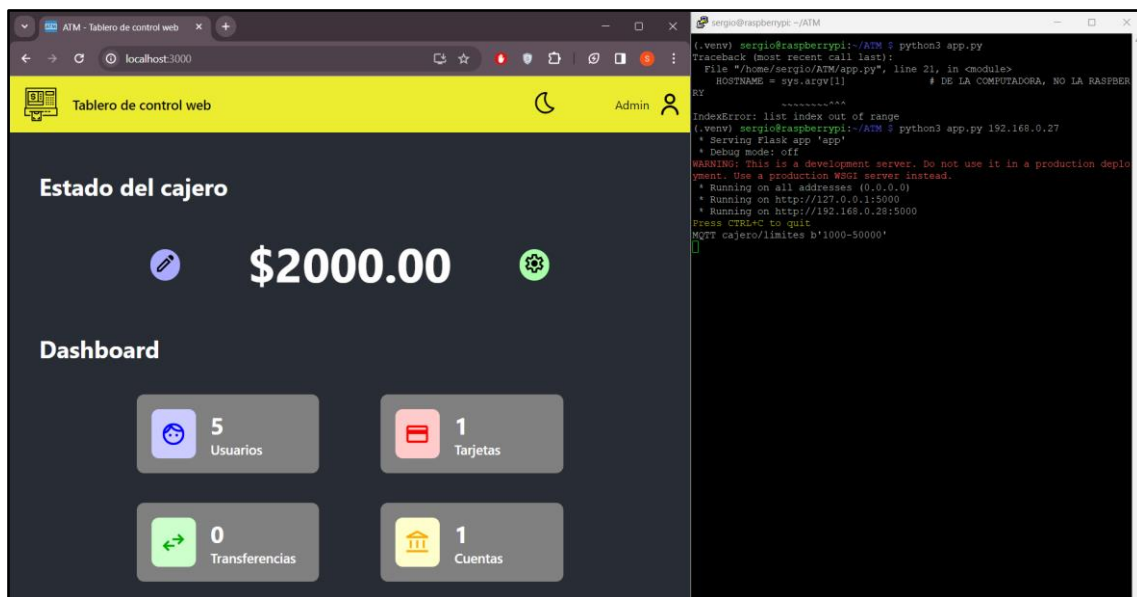


Figura 9.5. Visualización de efectivo en HomePage al iniciar cajero

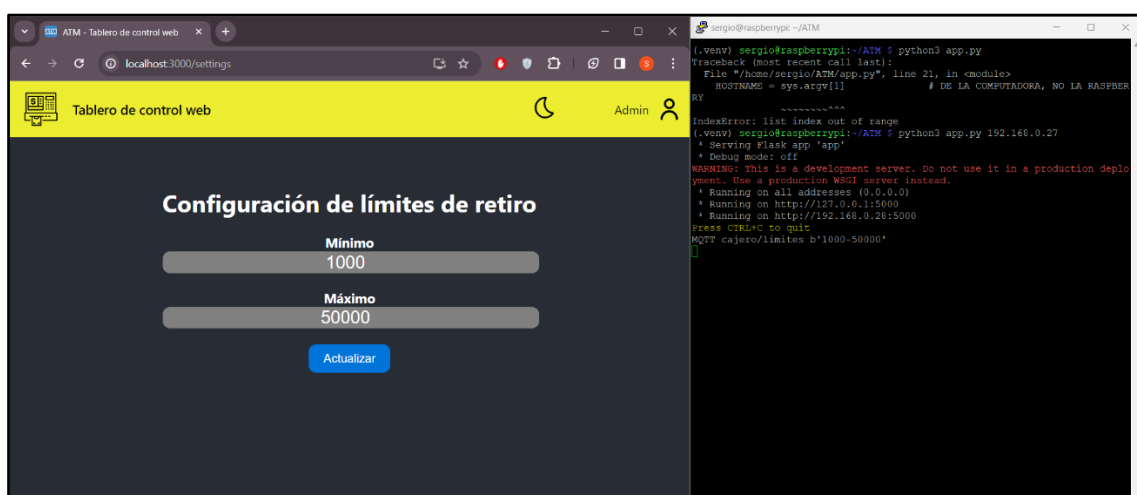
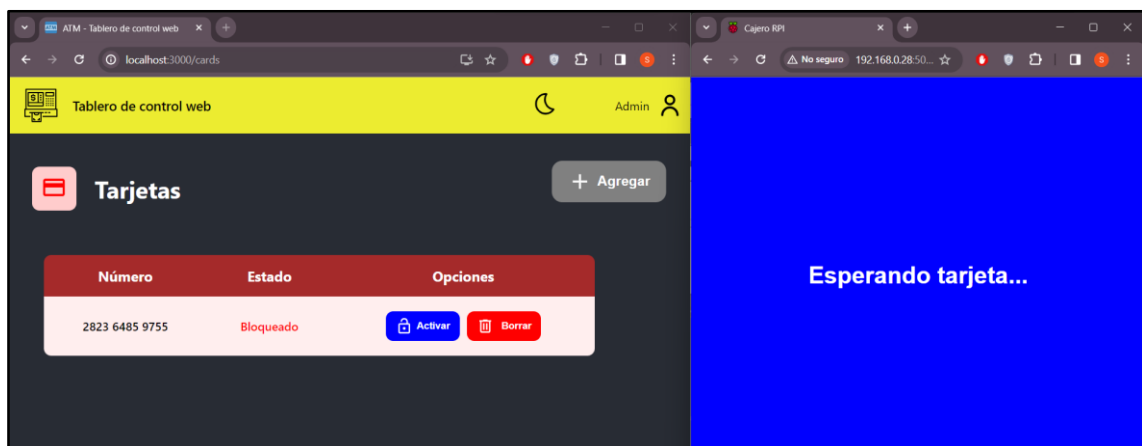
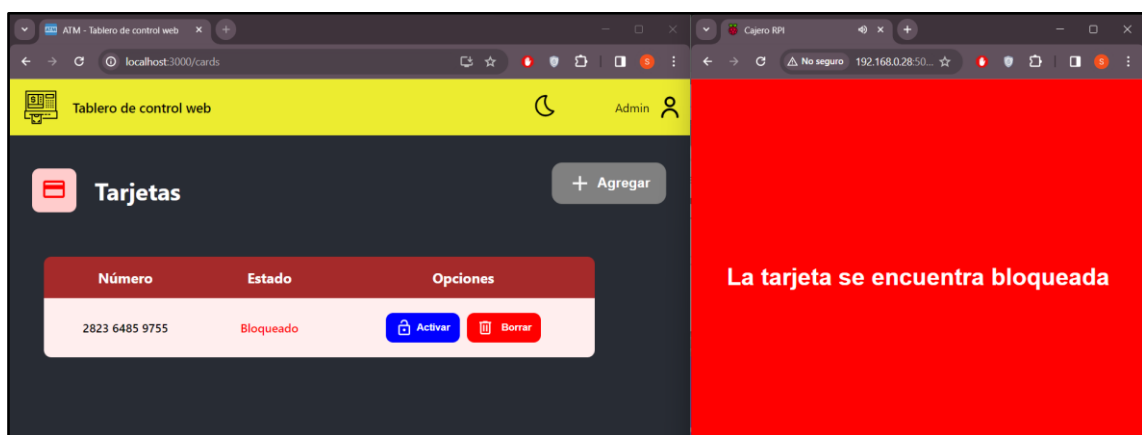


Figura 9.6. Carga de límites actuales en SettingsPage.

- ❖ **Autenticación completa:** el proceso completo de autenticación se testeó acercando en primer lugar la tarjeta registrada, pero bloqueando la misma desde el back-office. Luego, se desbloqueó la misma, y se accedió al estado de “Ingreso PIN”. Se realizaron entonces 3 intentos fallidos con un PIN incorrecto, y luego se volvió a leer la tarjeta para ingresar el PIN correcto (1235) y así poder acceder al menú de opciones. Video: <https://drive.google.com/file/d/15ta0TZ0bpmqcbkJvuTZ0G5MOAQa7PYT/view?usp=sharing>



*Figura 9.7. Tarjeta bloqueada y cajero en estado “Esperando tarjeta”*



*Figura 9.8. Tarjeta bloqueada y cajero en estado “Error” por dicho motivo*

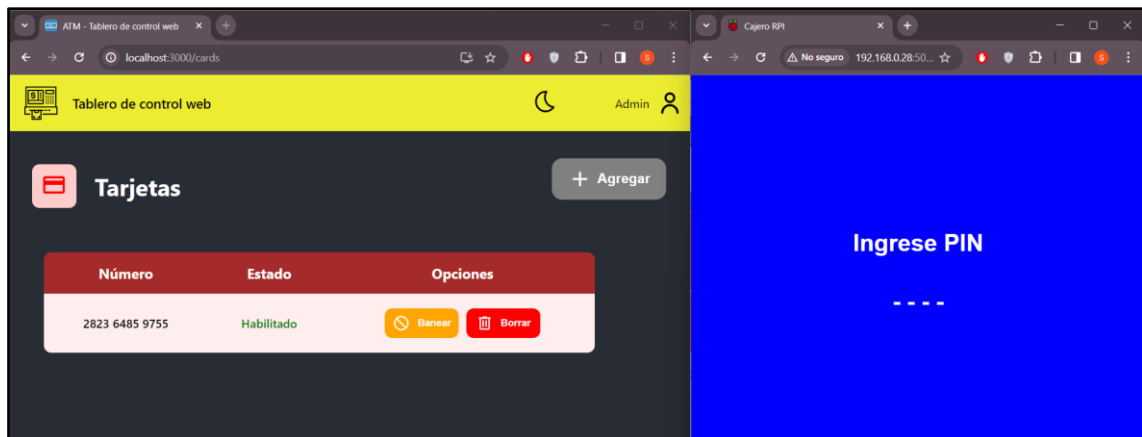


Figura 9.9. Tarjeta habilitada y cajero en estado “Ingreso PIN”

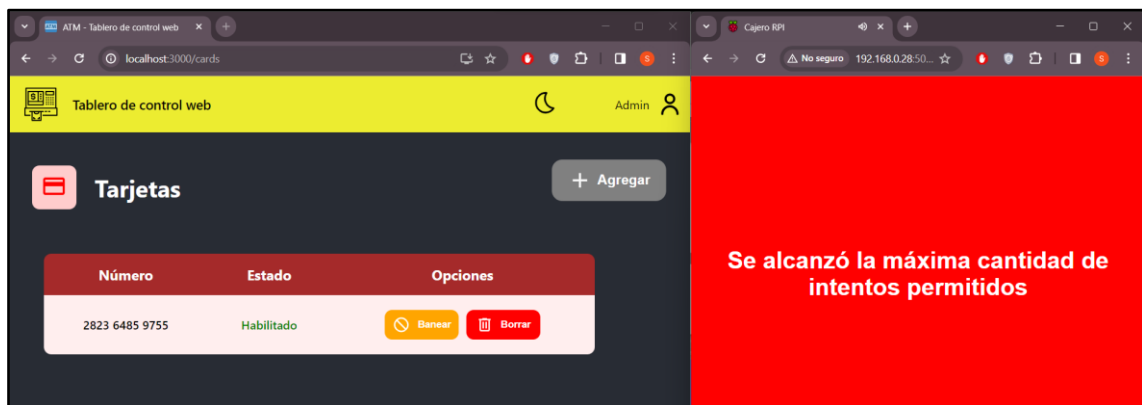


Figura 9.10. Cajero en estado “Error” luego de 3 ingresos erróneos de PIN

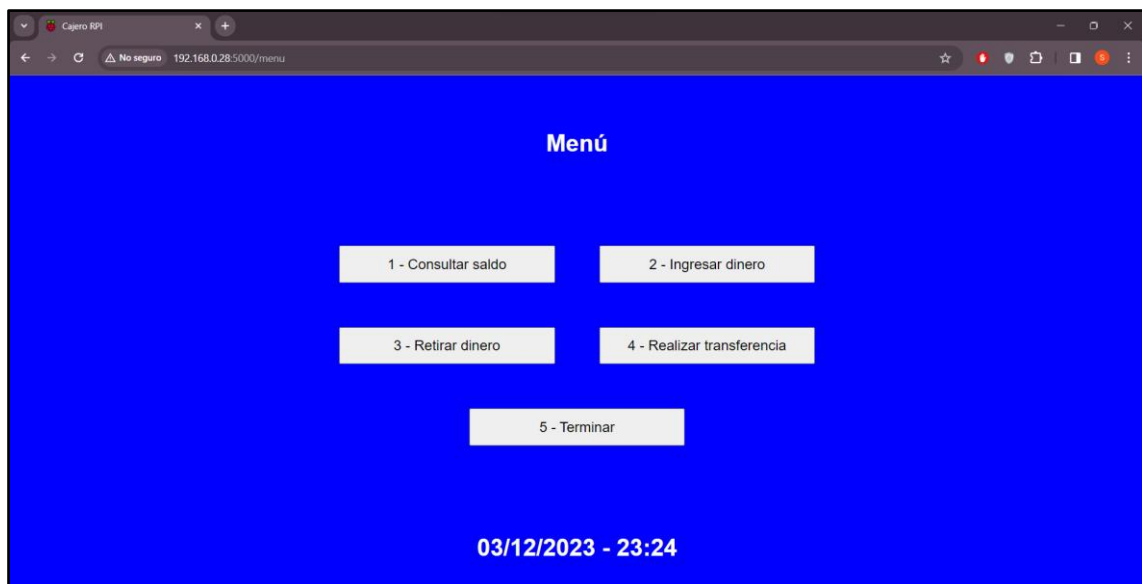


Figura 9.11. Vista del cajero para el estado “Menú”

- ❖ **Consulta de saldo y depósito:** para mostrar el funcionamiento de las opciones 1 y 2 del cajero, se partió de la situación de la cuenta con saldo cero, depositando \$1000 y \$1500 para mostrar la actualización del efectivo en el cajero y también el monto en la cuenta. Video: [https://drive.google.com/file/d/1ZSjfaHtOgjXFB2In20V2dhDMw\\_ul-p66/view?usp=sharing](https://drive.google.com/file/d/1ZSjfaHtOgjXFB2In20V2dhDMw_ul-p66/view?usp=sharing)

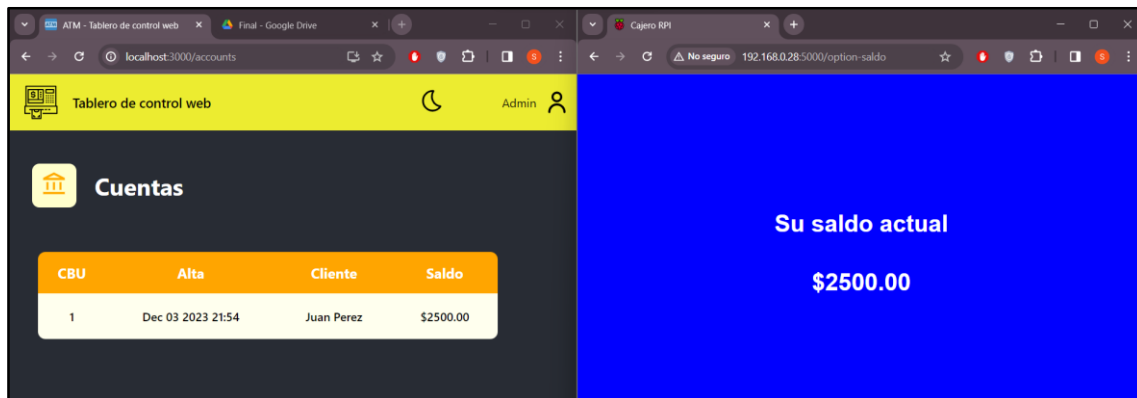


Figura 9.12. Vista de AccountListPage y cajero en estado "Muestra saldo"

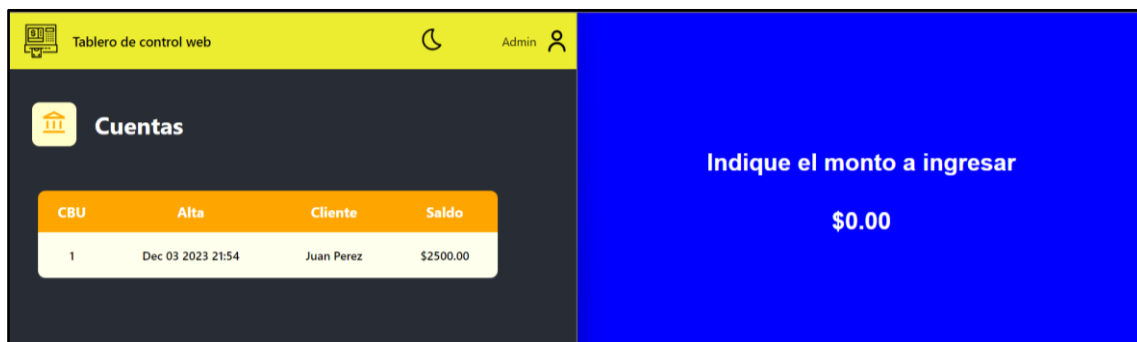
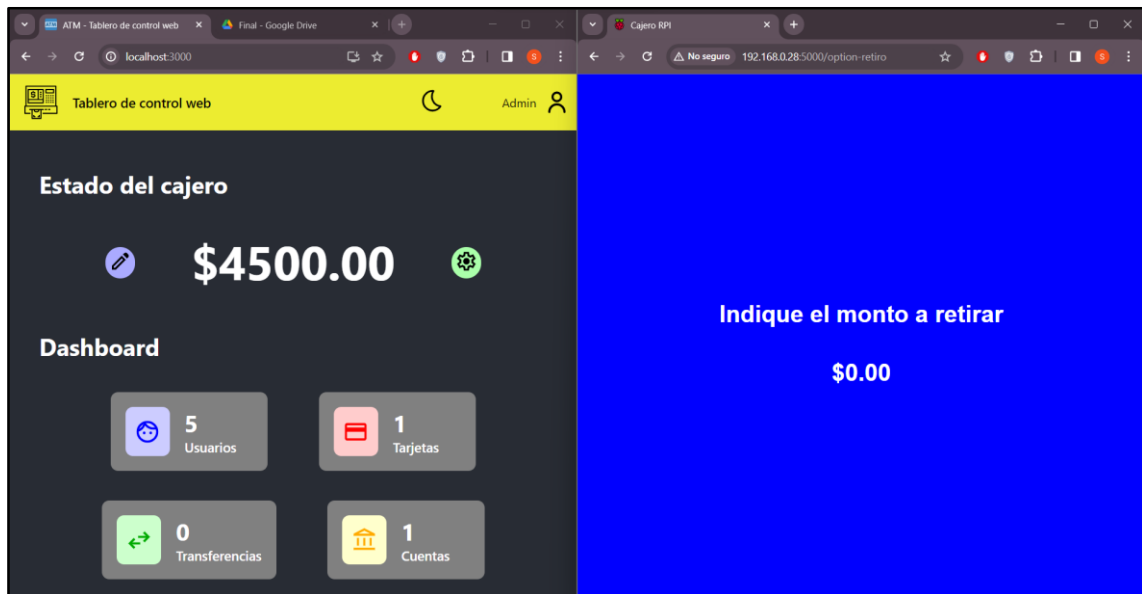


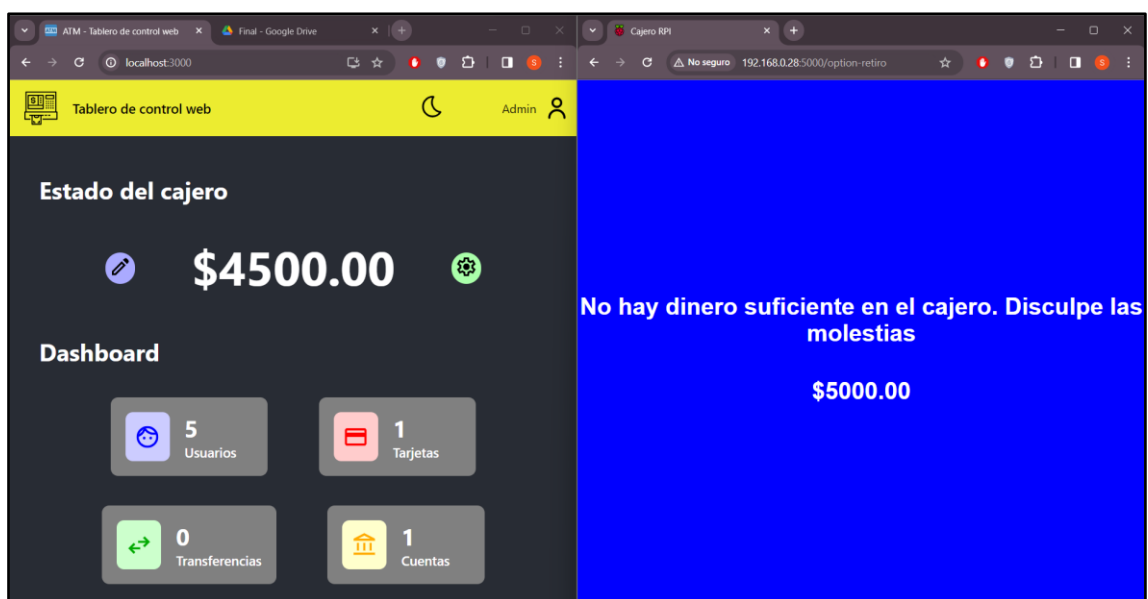
Figura 9.13. Vista de AccountListPage y cajero en estado "Ingreso dinero"

- ❖ **Actualización de límites:** desde la página de Opciones del back-office, se establecieron nuevos límites de extracción, intentando primero asignar valores restringidos por el sistema. La recepción de los límites en la Raspberry vía MQTT se observa por consola y en el archivo TXT. Video: [https://drive.google.com/file/d/1E45vXThOetFPiQigMD1LGDWVb\\_n-0Tvd/view?usp=sharing](https://drive.google.com/file/d/1E45vXThOetFPiQigMD1LGDWVb_n-0Tvd/view?usp=sharing)

- ❖ **Retiro de dinero:** se realizó una prueba exhaustiva de este proceso, tomando como situación inicial un cajero con \$4500 de efectivo y un saldo de \$2500 en la cuenta, cambiando los límites de extracción y se intentando indicar montos fuera de estas condiciones, para finalmente realizar un retiro de \$2000, actualizándose el efectivo y saldo. Video: <https://drive.google.com/file/d/1nKDVQkV1NweAjnxJtp50JDaHflfzLAHD>



*Figura 9.14. HomePage y cajero en estado “Retiro dinero”*



*Figura 9.15. Mensaje de dinero insuficiente en cajero al intentar retiro.*

- ❖ **Transferencia:** desde la interfaz del cajero, existiendo al menos 2 cuentas registradas en la base de datos (por lo que se debió registrar una nueva tarjeta en el back-office), se intentó indicar un CBU inexistente, el CBU de la propia cuenta origen y luego un CBU válido, probando transferir primero un monto que excede al actual (\$500) y luego uno menor. Video: <https://drive.google.com/file/d/1BnayWTI2tsK1i5RGMFh51JfnVtpD3H7E>

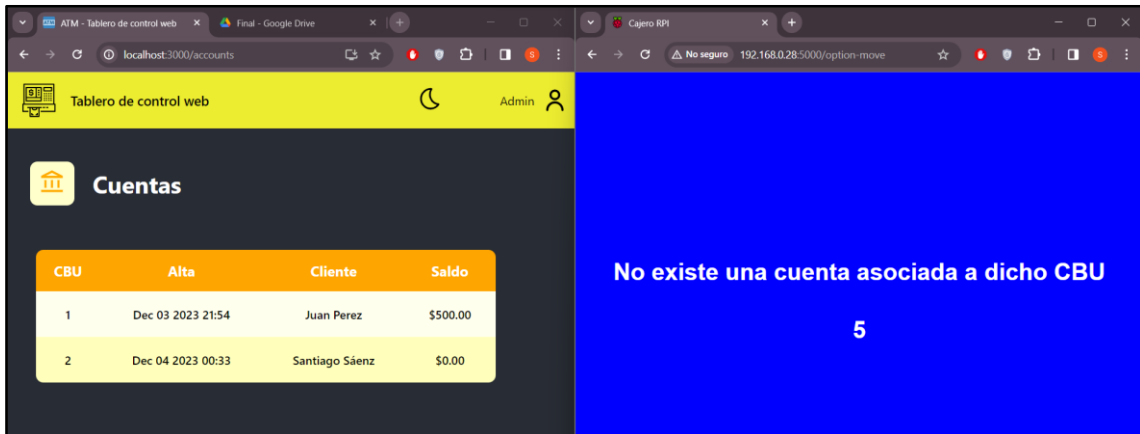


Figura 9.16. Estado “transferencia” mostrando error de CBU inexistente

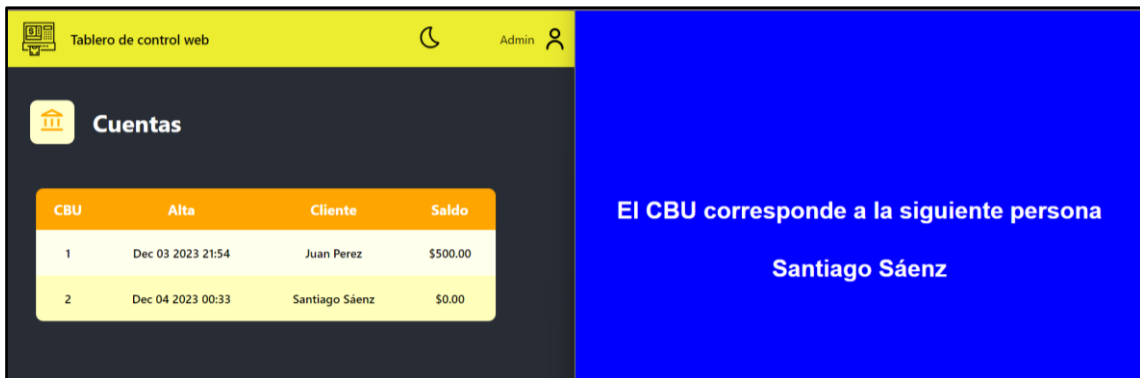


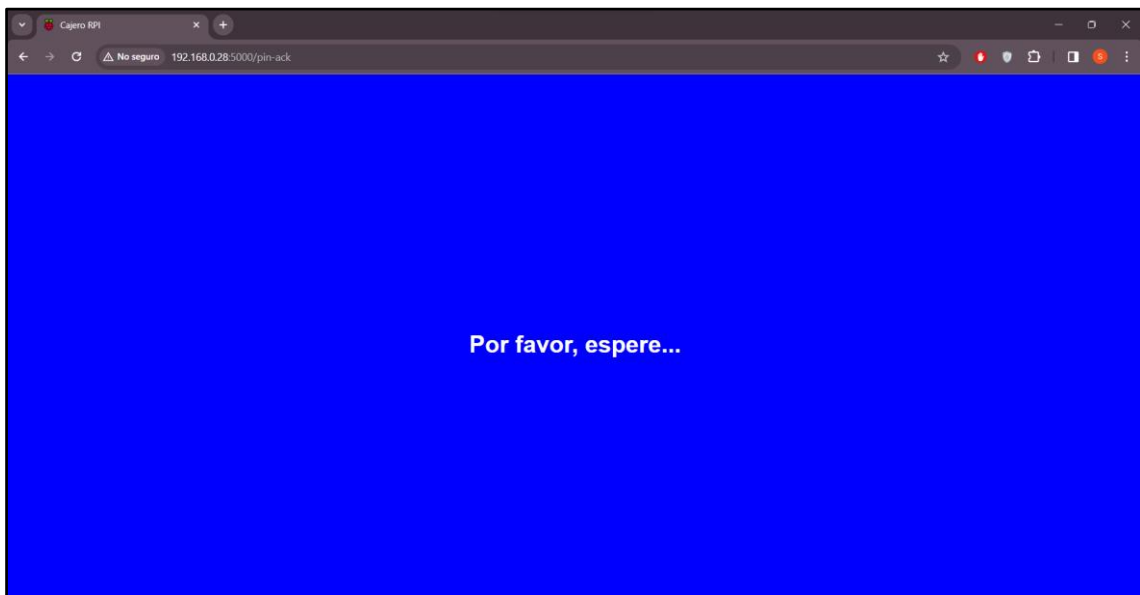
Figura 9.17. Estado “transferencia” mostrando datos del CBU destino “2”

Fecha	Origen	Destino	Monto
Dec 04 2023 00:34	Juan Perez	Santiago Sáenz	\$100.00

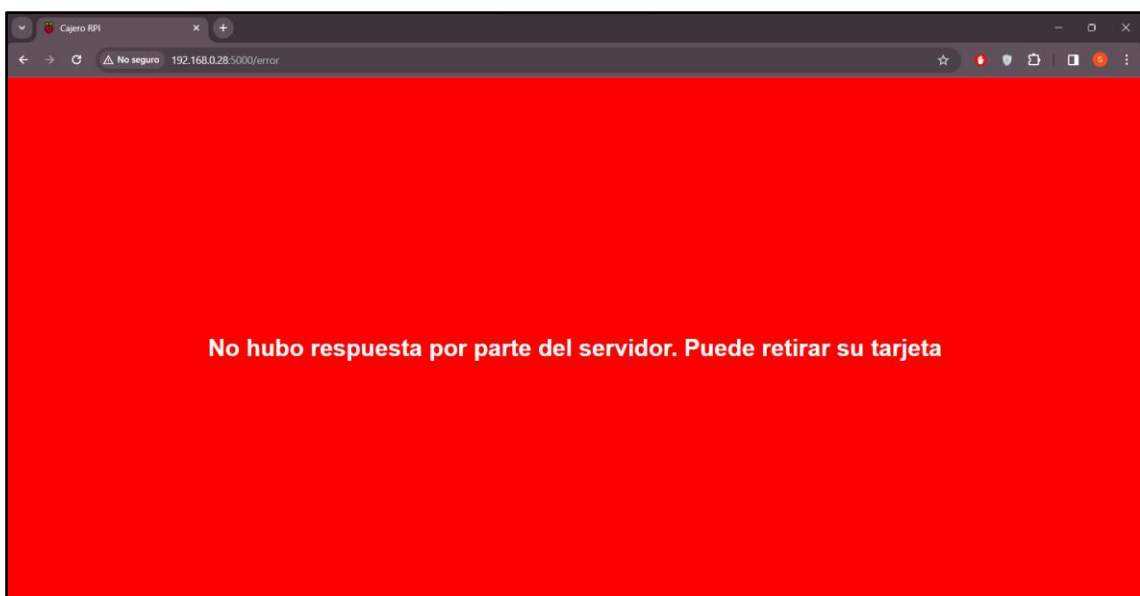
Figura 9.18. Transferencia creada, visible en MoveListPage



- ❖ **Caída del backend:** estando el cajero en el estado “Esperando tarjeta”, se desactivó el servidor backend del back-office vía CTRL+C, de modo que al leer una tarjeta por RFID e intentar conocer el PIN, no haya una respuesta y se deba mostrar un mensaje al usuario. Video: [https://drive.google.com/file/d/15uFG4a6L6A\\_2FnPDrVoCOrqTZ4bLi4I6](https://drive.google.com/file/d/15uFG4a6L6A_2FnPDrVoCOrqTZ4bLi4I6)

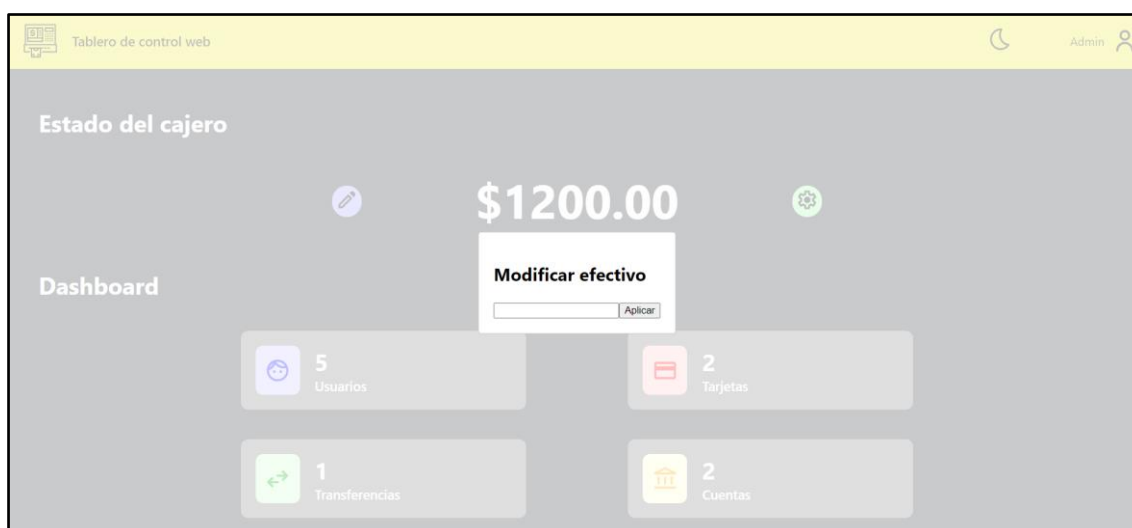


*Figura 9.19. Vista del cajero en el estado “Conociendo PIN”*



*Figura 9.20. Estado “Error” por ausencia de respuesta del backend*

- ❖ **Recuperación de preferencias:** al terminar la aplicación de Flask desde la consola de Raspbian, se guarda el valor del efectivo y los límites de extracción en archivos TXT, que son recuperados en la próxima sesión del programa. En caso de ser borrados, son creados nuevamente. Video: <https://drive.google.com/file/d/13PXqgWrzQdUBBwzr21cdaqLpTyypUeXF/view?usp=sharing>
- ❖ **Modificación del efectivo disponible:** desde el ícono de lápiz disponible en el HomePage, se accede a un modal para indicar el nuevo valor de efectivo. Al presionar “Aplicar”, se realiza el POST al backend, que a su vez realiza la publicación MQTT al cajero para que lo actualice. Video: <https://drive.google.com/file/d/1ZWOH3b90Bh9fdybeebeayAdnbjS6nGoN/view?usp=sharing>



*Figura 9.21. Modal de modificación de efectivo en HomePage*

Todo el código fuente se encuentra disponible en el siguiente repositorio público de GitHub: <https://github.com/tpII/2023-G3-ATM-RPIWeb>

Por otra parte, la bitácora con los avances descritos en cada semana desde la entrega del plan de proyecto se encuentra en la Wiki del repositorio: <https://github.com/tpII/2023-G3-ATM-RPIWeb/wiki/Bit%C3%A1cora>

## 10 – Bibliografía

- [1].BBC News Mundo (2017). *La curiosa historia de cómo nació el cajero automático*. Consultado el 2 de septiembre de 2023. Disponible en: <https://www.bbc.com/mundo/noticias-40417156>
  
- [2].Angulo, Francisco (2022). *El recuerdo del primer cajero automático en La Plata en 1986*. Disponible en: <https://www.laplata1.com/2022-05-13/el-recuerdo-del-primer-cajero-automatico-en-la-plata-en-1986-la-publicidad-de-la-abuelita-y-sus-duenos-estafadores-73689>
  
- [3].Manfredi, Melina (2021). *Invertir en un cajero automático: cómo es el negocio, cuánto hay que gastar y qué ganancia ofrece*. Disponible en: <https://tn.com.ar/economia/2021/10/16/invertir-en-un-cajero-automatico-como-es-el-negocio-cuanto-hay-que-gastar-y-que-ganancia-ofrece>
  
- [4].MongoDB (s.f). *MERN Stack Explained*. Consultado el 2 de septiembre de 2023. Disponible en: <https://www.mongodb.com/mern-stack>
  
- [5].Raspberry Tips (2022). *Instalar Raspberry Pi OS en Raspberry Pi*. Disponible en: <https://raspberrytips.es/instalar-raspbian-raspberry-pi/>
  
- [6].MDN Web Docs (2023). *Intercambio de recursos de origen cruzado (CORS)*. Consultado el 27 de septiembre de 2023. Disponible en: <https://developer.mozilla.org/es/docs/Web/HTTP/CORS>
  
- [7].Mongoose Docs (s.f). *Connections*. Consultado el 27 de septiembre de 2023. Disponible en: <https://mongoosejs.com/docs/connections.html>
  
- [8].GeeksForGeeks (2022). *Mongoose Aggregation.prototype.lookup() API*. Disponible en: <https://www.geeksforgeeks.org/mongoose-aggregate-prototype-lookup-api/>
  
- [9].MQTT.js (2023). Disponible en: <https://github.com/mqttjs/MQTT.js>

- [10]. Socket.IO (2023). *Server Initialization*. Consultado el 1 de octubre de 2023. Disponible en: <https://socket.io/docs/v4/server-initialization/>
- [11]. Axios HTTP (s.f). *La instancia Axios*. Disponible en: <https://axios-http.com/es/docs/instance>
- [12]. Ine4c Electronics (2021). *Instalación Broker Mosquitto en Windows*, p. 4-11. Consultado el 19 de octubre de 2023. Disponible en: <https://ine4celectronics.com/wp-content/uploads/2021/01/INSTALACION-BROKER-MOSQUITTO-WINDOWS-RASPBERRY.pdf>
- [13]. Trovó, Matteo (2023). *How to Use Wireshark for MQTT Analysis: An In-depth Guide*. Consultado el 19 de octubre de 2023. Disponible en: <https://cedalo.com/blog/wireshark-mqtt-guide>
- [14]. Ridout, Steve (2022). *MongoDB ObjectId Timestamp Converter*. Disponible en: <https://steveridout.com/mongo-object-time/>
- [15]. Light, Roger (2021). *Paho-MQTT Project Description*. Disponible en: <https://pypi.org/project/paho-mqtt>
- [16]. Python Packaging (2023). *Install packages in a virtual environment using pip and venv*. Consultado el 20 de noviembre de 2023. Disponible en: <https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/>
- [17]. Mercado Libre (2023). *Pendrive de 16 GB*. Disponible en: <https://www.mercadolibre.com.ar/pendrive-sandisk-cruzer-blade-16gb-20-negro-y-rojo/p/MLA6078534>
- [18]. Mercado Libre (2023). *Cable HDMI de 3 metros*. Disponible en: [https://articulo.mercadolibre.com.ar/MLA-709978022-cable-hdmi-mallado-3-mts-oro-1080p-4k-doble-filtro-\\_JM](https://articulo.mercadolibre.com.ar/MLA-709978022-cable-hdmi-mallado-3-mts-oro-1080p-4k-doble-filtro-_JM)

- [19]. Mercado Libre (2023). *Teclado numérico*. Disponible en: [https://articulo.mercadolibre.com.ar/MLA-1503753030-teclado-numerico-numpad-genius-\\_JM](https://articulo.mercadolibre.com.ar/MLA-1503753030-teclado-numerico-numpad-genius-_JM)
- [20]. Mercado Libre (2023). *Lector RFID Rc522*. Disponible en: [https://articulo.mercadolibre.com.ar/MLA-657877867-rfid-rc522-1356mhz-tarjeta-y-llavero-arduino-armodlrc522-\\_JM](https://articulo.mercadolibre.com.ar/MLA-657877867-rfid-rc522-1356mhz-tarjeta-y-llavero-arduino-armodlrc522-_JM)
- [21]. Mercado Libre (2023). *Raspberry Pi 3 Modelo B*. Disponible en: [https://articulo.mercadolibre.com.ar/MLA-703552293-raspberry-pi-3-rpi3-model-b-quad-core-12ghz-1gb-ram-\\_JM](https://articulo.mercadolibre.com.ar/MLA-703552293-raspberry-pi-3-rpi3-model-b-quad-core-12ghz-1gb-ram-_JM)
- [22]. React Community (s.f.). *React-Modal Documentation*. Disponible en: <https://reactcommunity.org/react-modal/>

## Apéndice

### A.1 – Partes de hardware

- Raspberry Pi 3, Modelo B.
- USB con el sistema operativo Raspbian [5]
- Cable HDMI, para visualización de consola
- Lector RFID Rc522

Componente	Precio en ARS	Ref.
Pendrive USB	\$5000	[17]
Cable HDMI	\$2200	[18]
Lector RFID Rc522	\$2200	[20]
Raspberry Pi 3 B	\$60000	[21]
<b>TOTAL</b>	<b>\$69400</b>	

*Precios actualizados a fecha de entrega final: 4 de diciembre de 2023*

Se pensó en utilizar un conversor HDMI a USB para poder visualizar la salida HDMI del Raspberry Pi en una notebook, pero solo se encontraron conversores con el propósito opuesto (USB a HDMI).

Otra consideración fue el uso de un teclado numérico o NumPad [19]. De esta forma, en la versión del cajero por consola, se limitaba la inserción de los caracteres '0' a '9' y la tecla Enter. Esta opción quedó descartada al reemplazarse por la aplicación de Flask, utilizándose el teclado de una notebook conectada a la interfaz mediante el navegador web.

### A.2 – Materiales recibidos

Del listado anterior se recibieron los siguientes componentes: Raspberry Pi 3 (modelo B), lector de tarjetas RFID Rc522 y un pendrive con Raspbian OS.



*Figura A.1. Materiales recibidos por la cátedra.*