VALENCIA COLLEGE

Department of Electrical & Computer Engineering Technology (ECET)
School of Engineering, Technology, and Advanced Manufacturing (ETAM)

CET 4370C
Advanced Programming Applications

PiLock

Performed by
Josué A. Bouchard
Submitted to
Dr. Ejaz

November 21, 2025

Abstract

PiLock is a low-cost NFC access control system designed to guard and log access to shared resources. Built on a Raspberry Pi and implemented in Python, it addresses the challenge of accountability in environments where resources such as golf cart keys, tools, or classroom equipment are used by multiple trusted individuals. The system has two primary functions: access control, allowing only whitelisted NFC tags to unlock the container, and logging, recording each borrowing and return event in a local database so that supervisors can review usage.

The design emphasizes affordability and modularity, using common components such as the RC522 NFC reader and a simple servo lock mechanism. A web interface provides managers with the ability to update permissions and view logs in real time from any device on the local network. By automating what is often a manual process, PiLock reduces reliance on supervisors, increases accountability, and creates a practical framework that can be adapted for a wide range of shared workspace scenarios.

Table of Contents

List of Figures

Chapter 1
Introduction

1.1 The problem

Shared workspaces pose a challenge when it comes to managing items: how do you control who uses a shared resource and ensure it is returned? Whether it's classroom markers and board erasers or the keys to a golf cart, tracking ownership and borrowing often requires a dedicated person, such as a manager, supervisor, or keeper, who may not be available at the exact moment you need the item in a hurry.

Another challenge is ensuring that individuals can access the resource promptly when needed. Time shouldn't be spent locating the appropriate person or completing administrative logs; rather, it should be directed toward accomplishing the task at hand.

1.2 The proposed solution

Whereas the current system relies on a human supervisor to manage keys, PiLock eliminates this bottleneck by providing automatic access and usage tracking, while still preserving oversight through centralized logs. The system will allow tags belonging to a whitelist to open the container, logging everything into a database, tracking every time the container is opened and closed. The main objective is to generate a sense of accountability and to allow higher-ups to track access.

This project is meant for cases where the people trying to access the resource are considered mostly trusted, meaning that a keeper would generally give them access to the resource if available, but accountability and logging are required.

1.3 Project structure

To this end, a Raspberry Pi was selected to host the system. As shown by Fig. 1.1, this first iteration of the project prioritized hardware simplicity and cost-effectiveness. An NFC reader is used to read tags used for authentication purposes, and a servo will work as a lock mechanism.

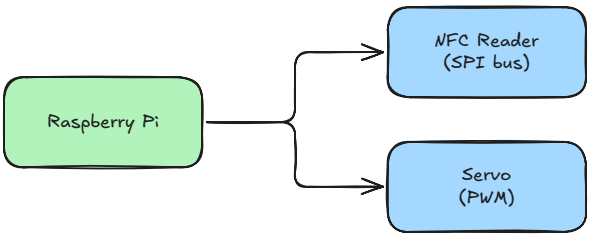


Fig. 1.1: Hardware block diagram.

The system was organized into two major components: the *Controller Software* and the *Manager Web Server*, as shown in Fig. 1.2.

The *Controller Software* serves as the intermediary between the Raspberry Pi's hardware and the *Manager Web Server*.

The *Manager Web Server* contains the *web interface* that managers can use to read the access logs and modify whitelisted tags. This module also exposes the *Controller API*, which handles tag validations and creates the access log records.

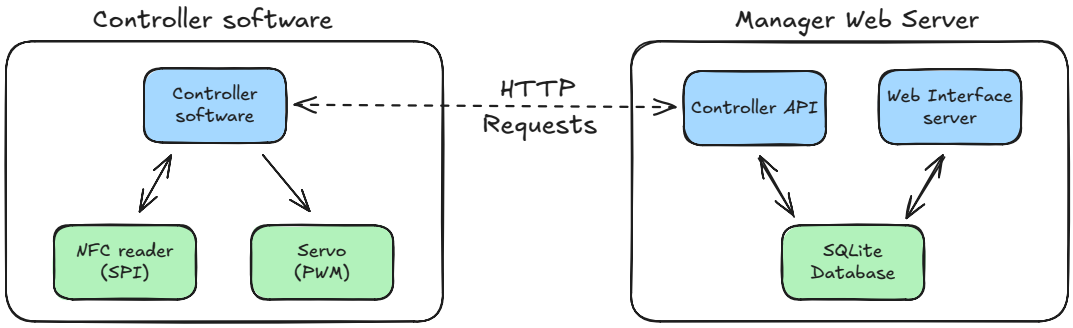


Fig. 1.2: Project structure diagram. Blue denotes Python programs; green denotes non-Python programs and hardware.

1.4 Case study

This project comes from the specific need of the Student Development office at Valencia College, where the golf cart keys are the resource to be guarded, and the most trusted employees are the student leaders. As a requirement from Valencia College, every use of the golf cart must be logged, from the moment the leader grabs the keys to the moment that they return them. In order to ensure that all forms are preserved, the supervisor takes the role of the keeper of the keys. But this didn't come without its issues.

Sometimes the supervisor would be remote, so she had to:

- Trust that we would know where the keys are
- Log personal information (VID) and access information (and time of aquisition)
- Grab the keys (resource)
- Return the keys
- Log the remnant of the information (time of return)

While the acquisition of the keys is usually not a problem, the time required to fill the logs, especially during time-pressure moments such as running events, makes it that sometimes errors are committed.

PiLock seeks to replace protocols with an automated system that provides oversight through access logs while still facilitating access to the resources inside the container.

1.5 Similar projects

While the hardware part is similar in several studies, the scope, price, and especially the degree of sophistication of its software will set this project apart from others. To ensure a fair and equivalent comparison, only projects involving low-to-moderate cost and small scale were analyzed. A couple of examples include:

- A study from India shows how they built an Internet of Things (IoT) smart locker [1]. While this paper is the closest thing to this project, what will set PiLock apart is its standalone nature and the degree of sophistication of its software.
- Example video tutorials exist on how to make a smart lock, such as [2]. In this case, the circuit of the video uses Bluetooth instead of Wi-Fi. While the decision for using Wi-Fi is explained later, this project proposes to expose a useful UI with modifiable access control that doesn't require reflashing the controller.

Chapter 2
Hardware and Software

2.1 Hardware

2.1.1 The Raspberry Pi

The Raspberry Pi 4 model B (see Fig. 2.1) was chosen for its flexibility, relatively low cost, silence, energy efficiency, and its ability to [3]:

- Exposing General Purpose Input/Output (GPIO) pins that can be used to interact with external sensors and actuators.
- Be able to interface with SPI devices.
- Have network capabilities.
- Run a Linux-based operating system that allows the execution of CPython code and dependencies, alongside a SQLite database.

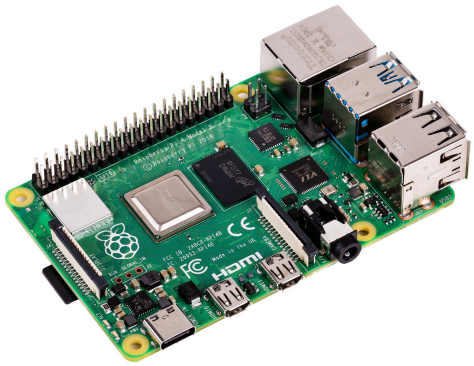


Fig. 2.1: A Raspberry Pi 4B

The Raspberry Pi has three primary tasks:

- Interface with the NFC reader and container opener mechanism and execute logic to determine whether the container should be opened or not.
- Host the SQLite database.
- Expose the access log stored in the SQLite database via a web application with authentication.

2.1.2 The NFC reader

The decision of using NFC instead of other alternatives, such as barcodes or PINs, comes from a security standpoint: it's easier to copy/screenshot a barcode or make one if the information in the barcode is known [4]. Furthermore, NFC tags become a "Something you have" authenticator [5]. That means that if a person has their tag stolen, that should be quickly identifiable, and easily informed so that the privileges are revoked.

This first iteration uses a reader based on the RC522 [6] with a MIFARE Classic 1K card, as shown in Fig. 2.2. The RC522 is an ISO/IEC 14443 A/MIFARE reader writer, which means is only compatible with that specific type of NFC tag.

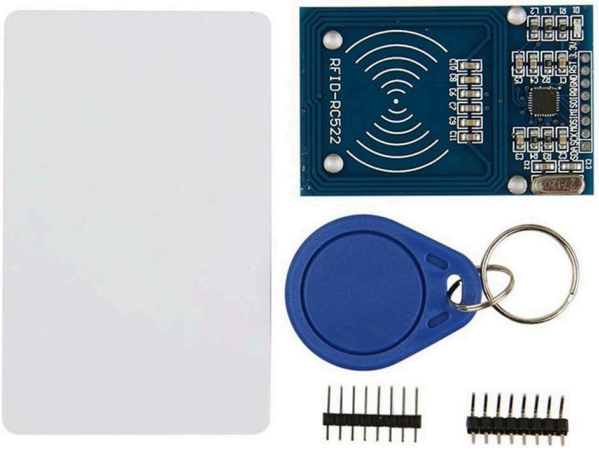


Fig. 2.2: An RC522 NFC reader with a MIFARE Classic 1K card and key ring

A technical tradeoff taken in the selection of the tags is their built-in security capabilities vs cost. ISO/IEC 14443 A/MIFARE tags are known for their availability and low cost (less than 1$ per tag), while alternatives that include encryption, such as the MIFARE DESFire EV3, tend to cost considerably more (more than 2$ per tag) [7], [8]. A similar thing happens with the reader, which could be replaced by an NXP PN5180 in order to read the DESFire tags.

While, in this case, the major factor was availability of the components and information, it can be noted that the modular structure of the PiLock enables for an easy switch of the reader and tag.

2.1.3 The locking mechanism

For this first iteration, an SG90 servomotor, such as the one in Fig. 2.3, is used to close and open a latch, allowing the person to open or close the lid.

Fig. 2.3: SG90 micro servomotor.

As it will be explained later, the `ServoLock` abstraction allows for easily swapping the logic in the software part. That enables a simple exchange of the servomotor as a locking mechanism for another component, such as a keyed solenoid latch, which allows for a good fail-safe and operation under power-off.

The SG90 servomotor was chosen due to its price and low power requirements. While there is no official datasheet as it's made by several Chinese companies, after consulting several online forums, the following table can be compiled:

Table 2.1
SG90 servomotor maximum values
(rough approximation)

	min	max
Voltage	4.8 V	6.0 V
Current (Idle)	10 mA	Unknown
Current (Running - light load)	100 mA	250 mA
Current (Stall)	360 mA	700 mA

According to the Raspberry Pi 4 Model B datasheet, the board is powered from a 5 V, 3 A USB-C supply, and the downstream USB current is limited to approximately 1.1 A [9]. The 5 V header pins are connected directly to the 5 V input rail, so the current available to external devices depends on the power supply rating and the Pi's own consumption. Under typical conditions, the Pi itself can draw on the order of 1.0-1.5 A, leaving roughly 0.5-1.0 A available for external 5 V peripherals when using the official 3 A power supply. Given the SG90′s estimated stall current of up to 700 mA from Table 2.1, this is sufficient for a single servo in this prototype, though a dedicated 5 V supply for actuators would be preferable in future iterations to reduce noise and voltage dip on the Pi's power rail.

2.2 The software

2.2.1 Python

Python was selected as the programming language for this project because of its rapid development cycle, readability, and extensive ecosystem of libraries for solving practical problems. Its versatility is illustrated by Meta's use of Python to launch Threads in just five months [10].

Modern Python features (>3.10) were used in this project, especially around the topic of type hinting [11], [12]. This allows "[…] easier static analysis and refactoring, potential runtime type checking, and (perhaps, in some contexts) code generation utilizing type information" [11]. in this project, basedpyright was used as a static analyzer devtool to improve developer experience, aid in the detection of bugs, and facilitate code navigation [13].

The project uses FastAPI extensively, which combines strong performance with modern features such as type hinting (reducing bugs) and automatic generation of an OpenAPI schema [14]. Together with supporting libraries, such as httpx for web requests, Jinja for templating, Pydantic for schema validation, and SQLModel for database interactions, this stack provides a reliable foundation for building a web server while maintaining a rapid pace of development.

Other libraries used in this project include structlog for structured logs, gpiozero for controlling the servos, pi-rc522 for communicating with the RC522 NFC scanner, and uvicorn, a high-performance, Asynchronous Server Gateway Interface (ASGI) web server implementation for Python.

2.2.2 UV

Given that Raspberry Pi strongly encourages developers to utilize virtual environments instead of installing Python packages globally, it was decided to use Astral's uv, "An extremely fast Python package and project manager, written in Rust" [15].

It was selected due to its easy install process, friendly interface, and speed.

2.2.3 PiGPIO

In order to use PWM using hardware clocks, it's necessary to talk directly to the BCM2711 chip used by the Raspberry Pi. However, because the system runs in user space on top of an operating system, it needs some piece of software that can actually communicate with the IC. The Linux kernel exposes `/dev/mem`, which is used by PiGPIO and it's daemon pigpiod to interface with the PWM peripherals. It's used by `gpiozero`.

2.2.4 Systemd

Another consequence of the Raspberry Pi software not being in a bare-metal environment (there is an OS present) is that the operating system does not run the code we write automatically. In order to solve this, Raspberry Pi OS comes with Systemd, a service manager for Linux that starts, monitors, and coordinates system processes in a parallel and efficient way.

Each service is defined in a `.service` file, and Systemd knows how to run them in the correct order based on the service definition. This ensures the Controller Software and Manager Web Server start automatically on boot and recover from crashes.

The services written for this project can be found in Figs A.1, A.2, and A.3.

2.2.5 Journald

As an extra layer of observability, logs can be issued from the modules. The logs generated by services run by Systemd piped to the standard output are automatically writen into Journald, a Linux built-in system that collects, stores, and indexes logs from the kernel, system services, and user applications in a structured, binary format.

Chapter 3
Contributions

3.1 Overview

The PiLock software, as shown in Fig. 1.2, was deliberately divided into two modules: the Controller Software and the Manager Web Server. This architecture, following the Single Responsibility Principle, reduces coupling and allows each part to be developed independently [16].

This modularity also allows for future scalability. One of the requirements for this project was not to depend on any external system, so both modules are run on the same device. However, the system was built in such a way that an assumption is not made. This allows the Manager Web Server to be moved to a cloud environment with very little architectural changes, enabling the Raspberry Pi to be replaced by a more specialized, low-power microcontroller dedicated solely to hardware control. The only requirement it has is that the environment must be able to run CPython.

Regarding the physical connections from the Raspberry Pi to the peripherals, they were made according to Table 3.2. While a perforated board could have been used, it was determined that its benefits did not outweigh the additional complexity, so standard jumper wires were used. If future iterations were to require more complex circuitry (such as extra SPI devices or more peripherals), then the idea could be revisited.

Table 3.2
Raspberry Pi connection table

Physical pin numbering (Raspberry Pi)	**Device**	**Device Pin**
35 (PCM_FS)	Servo	PWM signal
34 (GND)	Servo	GND
24 (GPIO 8)	NFC	SDA
23 (SCLK)	NFC	SCK
22 (GPIO 25)	NFC	RST
21 (MISO)	NFC	MISO
20 (GND)	NFC	GND
19 (MOSI)	NFC	MOSI
17 (3V3)	NFC	3.3V
2 (5V)	Servo	5V

3.2 Manager Web Server

This module exposes two servers, the Controller API and the Web Interface Server. As seen in Fig. 1.2, both servers utilize the same database. While that could be seen as troublesome, since that might strengthen the coupling, the ownership of data models and the use of read-only models was chosen to get transactional consistency, while keeping the system mostly uncoupled. In this design, only the Controller API is allowed to modify the Tags and Access Logs tables, while the Web Interface Server interacts with read-only representations of the same data, as it can be seen in Fig. 3.1.

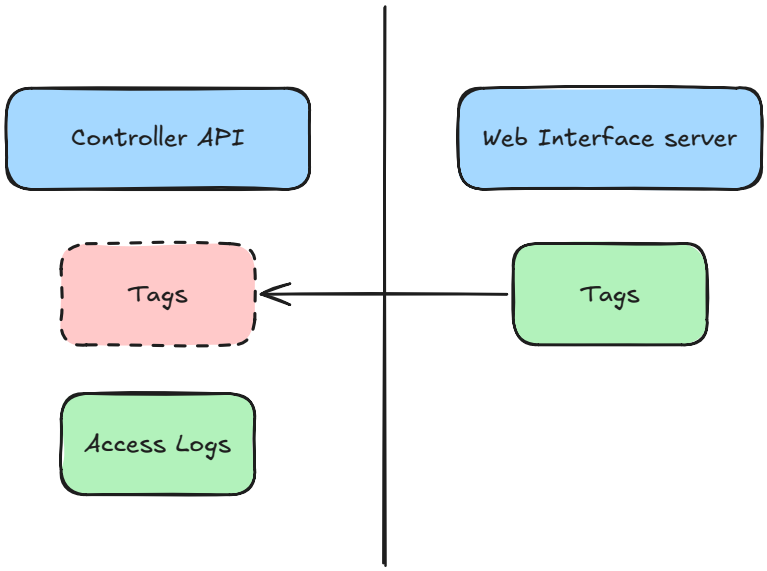


Fig. 3.1: Manager Web Server data models (green: owned, red: read-only)

All the data models from Fig. 3.1 are stored in an SQLite database due to its simplicity: it's composed of a single database file and requires no daemon to read it or write to it.

3.2.1 Web Interface Server

The Web Interface Server is the place where management can see, create, and edit the tags that can open and close the container (Fig. B.1, Fig. B.2, and Fig. B.3) and see the access logs and download them (Fig. B.4 and Fig. B.5). It's exposed to the public, and only responds with HTML.

This server is built using an architecture similar to a Model-View-Controller (MVC), as shown in Fig. 3.2.

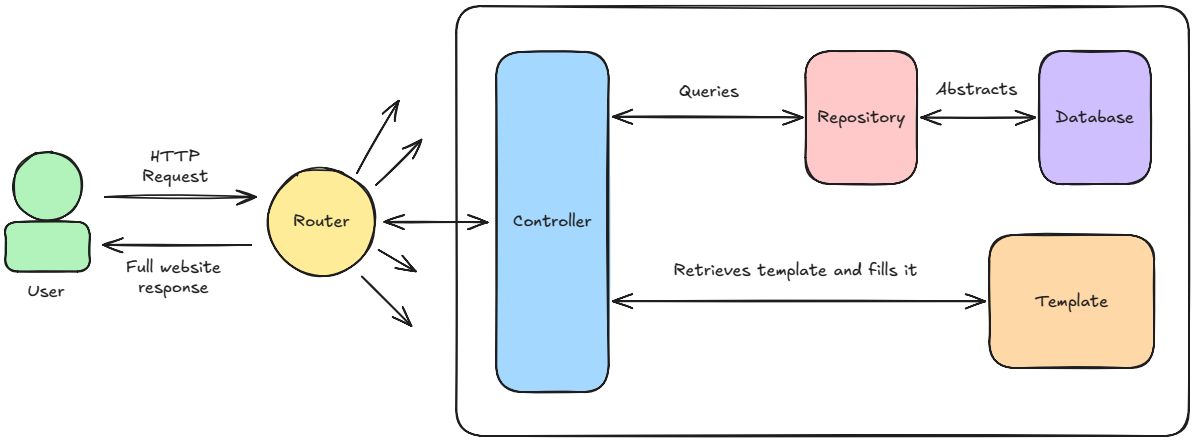


Fig. 3.2: Representation of the MVC architecture

In short, when a user requests a resource, such as listing all tags, the router calls the right controller, which runs most of the logic. The controller logic usually consists of performing some operation over entities using the functions provided in the repository, and using the data to fill a template and return it to the user.

In this context, and as shown in Fig. 3.2, the repository is a set of functions that seek to hide how the database operates internally, exposing entities such as tags (Listing C.4) or access log (Listing C.5) records. That means that if in the future one of the database models wants to change, or the entire database wants to be substituted by another one, such as Postgres, all the changes can be done in a centralized fashion on the repository.

3.2.2 Controller API

The Controller API is meant to be an HTTP REST api that sends and receives information using JSON. It's meant to be called not by users, but by the Controller Software.

As seen in Fig. 3.1, this server is responsible for creating the access logs, both for successful and unsuccessful attempts of locking or unlocking the container. It exposes a single endpoint as it can be seen in Listing C.6, whose logic can be summarized in Fig. 3.3.

In this first iteration, utilizing the assumption that both the Controlling Software and the Manager Web Server are operating in the same machine, the only security measure taken is not to expose the server to the internet. In case this module was decided to be moved to a cloud provider, some security mechanism would have to be implemented, such as mutual TLS (mTLS) or authentication keys.

3.3 The Controller Software

The Controller Software is the piece of code that interfaces with the Raspberry Pi sensors and actuators. As previously mentioned in Section 3.1, it's the only portion of code that is tied to the hardware it's running on.

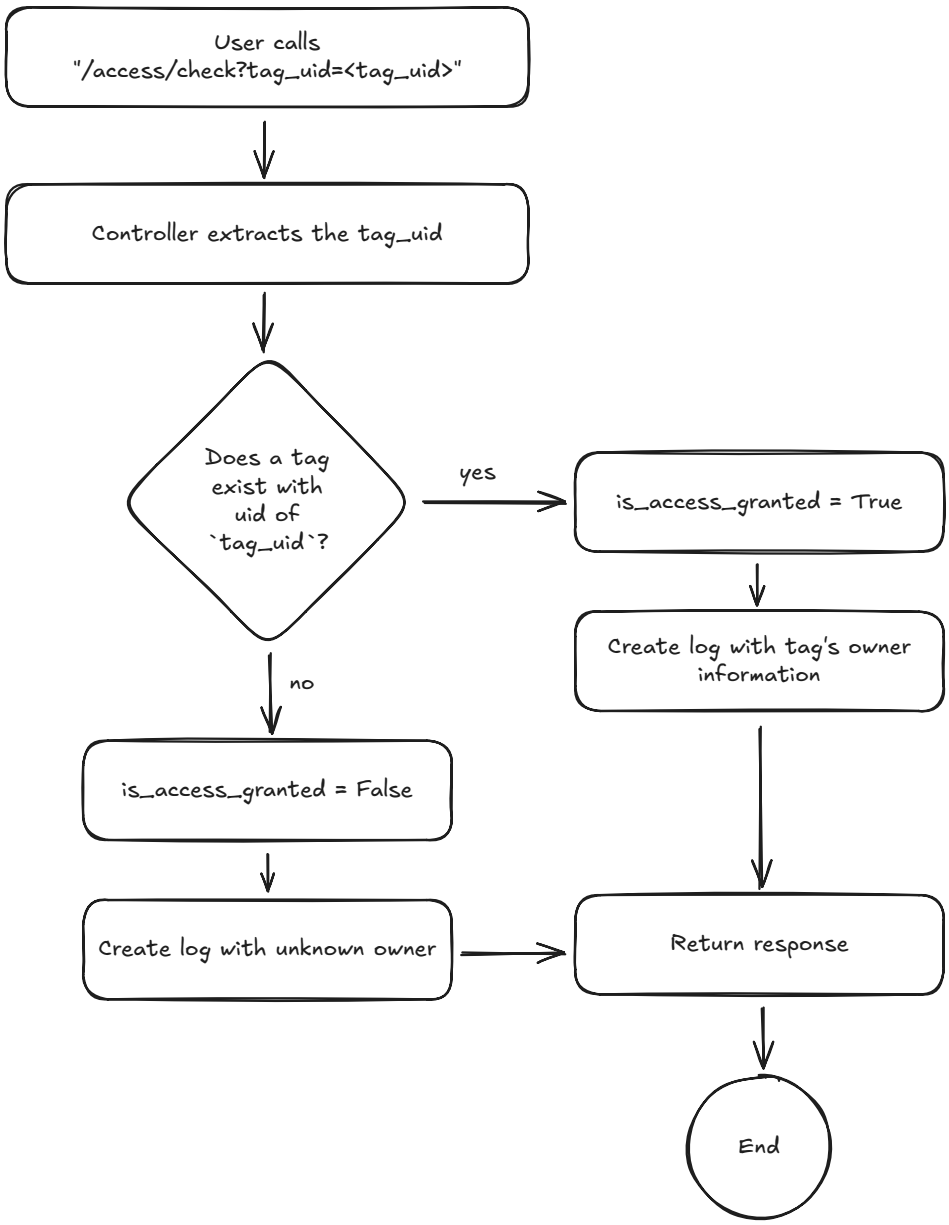


Fig. 3.3: Flowchart of Controller API logic

A simplified version of its behavior can be found in Fig. 3.4. While the real code includes things such as extra logging and time-based debouncing. As seen on Listing C.7, it first polls the NFC reader to check whether there is an NFC tag present or not (Listing C.8). Given that multiple NFC tags could respond at the same time to the reader, an anticollision step is performed to isolate only one tag. If it succeeds, the UID of the tag is used as the tag identifier and sent to the server to validate authorization, as it can be seen on Listing C.9. If it passes the validation, the lock is toggled and access is given. If any error happens in the middle, access is not granted, and the source of the error is logged into journald.

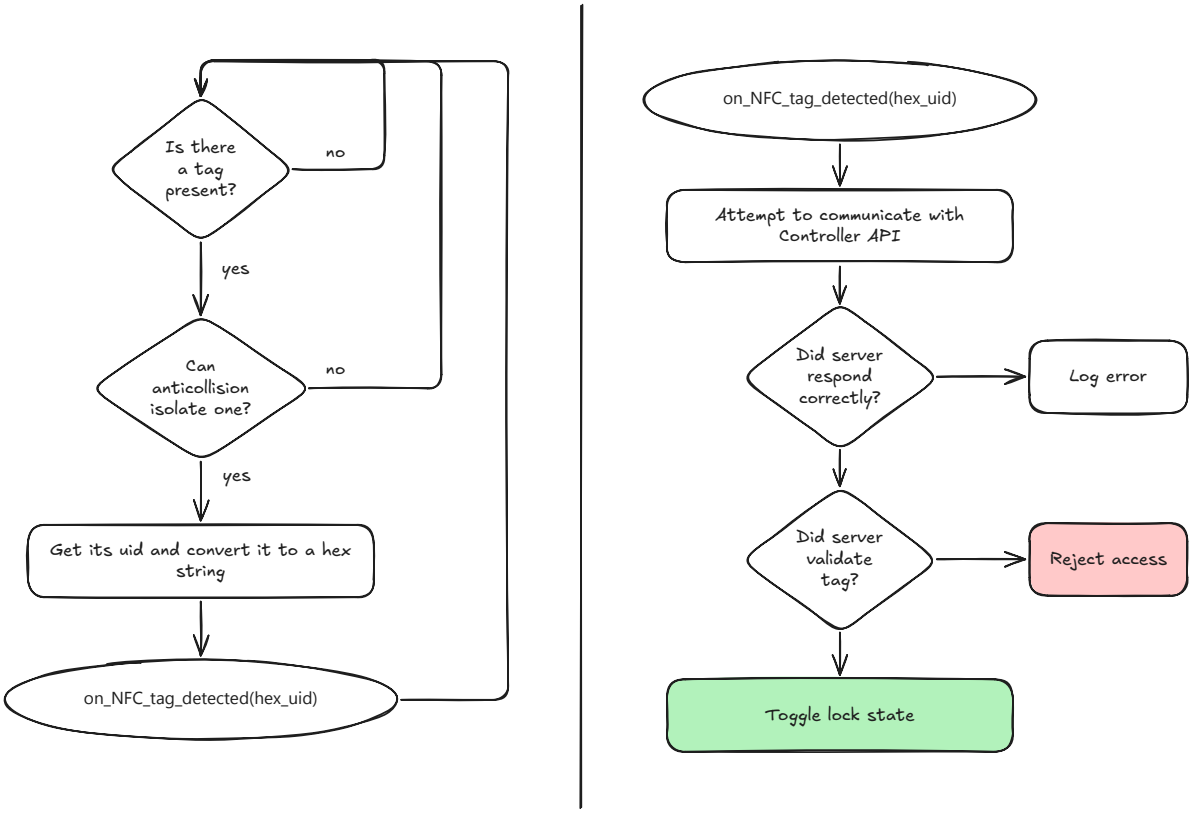


Fig. 3.4: Flowchart of Controller Software main behavior

3.4 Future improvements

While several improvement opportunities have been discussed in this report. There are some ideas that weren't able to be implemented, due to complexity, timing, or price.

Authentication could be implemented using cookie-based authentication, contrasted to common JWT authentication methods, based on a comparison of [17] and an analysis of the considerations from [18]. Passwords would be hashed in accordance with the recommendations in [19], and stored in a database. Alternatively, a third-party authentication provide could be used to ease that.

A screen that turns on and goes to sleep when not needed could be implemented for better communicating the state of the system with the user.

If the requirement of a self-standing product is dropped, or if the opportunity to ofload some of its work to an external system is allowed, the Raspberry Pi could be exchanged for a smaller, more specialized, less expensive and power consuming microcontroller, such as an ESP32.

Finally, the enclosing also could use a remodeling. While for this case a generic box was used, a custom built version could be made in 3D and either laser cut it in wood or plastic and assemble it, or 3D print it.

Chapter 4
Conclusion

PiLock addresses a practical problem faced in shared workspaces: ensuring that valuable resources can be accessed quickly by trusted users while still maintaining accountability. By integrating an NFC reader, a simple servo-based lock, and a Raspberry Pi running a Python web application, the system provides a balance between accessibility and oversight. Key functions such as access control, logging, and return tracking are combined with a user-friendly web interface for managers, ensuring that the system is both functional and easy to administer.

While the current design emphasizes affordability and proof-of-concept implementation, PiLock demonstrates how relatively simple hardware and open-source software can replace manual supervision with an automated, reliable process. Future improvements could include support for more secure NFC standards, encrypted communication between modules, or offloading the computing side to the cloud for centralized monitoring.

References

[1] C. Bharatiraja, P. Chittoor, and Y. Bhargava, “An IoT based centralized smart locker using RFID technology,” in *AIP Conference Proceedings*, Feb. 2023, p. 20098. doi: 10.1063/5.0101139.

[2] “How to Make Mobile Control Door Lock at Home.” [Online]. Available: https://www.youtube.com/watch?v=fWzBxd8Nwco

[3] “Raspberry Pi 4 model B.” Accessed: Sept. 29, 2025. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

[4] P. Tomes, “Barcode vs. NFC Wallet Passes.” Accessed: Sept. 29, 2025. [Online]. Available: https://help.passkit.com/en/articles/12395803-barcode-vs-nfc-wallet-passes

[5] D. Temoshok *et al.*, “Digital Identity Guidelines,” Gaithersburg, MD, technical report NIST SP 800-63-4, Aug. 2025. doi: 10.6028/NIST.SP.800-63-4.

[6] NXP Semiconductors, “MFRC522 - Standard performance MIFARE and NTAG frontend,” Apr. 2016. [Online]. Available: https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf

[7] “20 PCS BABIQT MIFARE Classic 1K Tags S50 RFID Sticker NFC Tags Smart Mifare Tags Stickers RFID ISO 14443A 13.56 MHz Mifare MF1 Stickers Self-Adhesive Commercial Access Proximity Cards 1 inch Round : Cell Phones & Accessories.” Accessed: Nov. 20, 2025. [Online]. Available: https://www.amazon.com/BABIQT-Sticker-Classic-Self-Adhesive-Stickers/dp/B0BZRS35CT

[8] “1-Pack Mi-fare Desfire EV3 NFC Cards (2K/4K/8K Options) ISO/IEC 14443A Compliant : Industrial & Scientific.” Accessed: Nov. 20, 2025. [Online]. Available: https://www.amazon.com/1-Pack-Mi-fare-Desfire-Cards-Options/dp/B0F2GY5TTD

[9] “Datasheet - Raspberry Pi 4 Model B.” Accessed: Nov. 19, 2025. [Online]. Available: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf

[10] “QCon London: Meta Used Monolithic Architecture to Ship Threads in Only Five Months.” Accessed: Sept. 30, 2025. [Online]. Available: https://www.infoq.com/news/2024/04/meta-threads-instagram-5-months/

[11] G. van Rossum, J. Lehtosalo, and Ł. Langa, “PEP 484 - Type Hints.” Accessed: Nov. 21, 2025. [Online]. Available: https://peps.python.org/pep-0484/

[12] P. PRADOS and M. Moss, “PEP 604 - Allow writing union types as X \vert{} Y.” Accessed: Nov. 21, 2025. [Online]. Available: https://peps.python.org/pep-0604/

[13] [Online]. Available: https://docs.basedpyright.com/latest/

[14] “FastAPI.” Accessed: Sept. 30, 2025. [Online]. Available: https://fastapi.tiangolo.com/

[15] Astral, “uv.” [Online]. Available: https://docs.astral.sh/uv/

[16] Oloruntoba, Samuel, Walia, Anish Singh, and Kurup, Manikandan, “SOLID Design Principles Explained: Building Better Software Architecture.” Accessed: Nov. 19, 2025. [Online]. Available: https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

[17] “JWTs vs. sessions: which authentication approach is right for you?.” Accessed: Sept. 30, 2025. [Online]. Available: https://stytch.com/blog/jwts-vs-sessions-which-is-right-for-you/

[18] “JSON Web Token for Java.” Accessed: Sept. 30, 2025. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html

[19] “Password Storage.” Accessed: Sept. 30, 2025. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

Appendix A - Systemd Service Definitions

ini

controller-api.service

```ini
[Unit]
Description=Controller-api
After=network.target network-online.target
Wants=network-online.target

[Service]
Type=simple
User=cotuex
Group=cotuex

WorkingDirectory=/home/cotuex/pilock/manager-web-server
Environment=PYTHONUNBUFFERED=1
ExecStart=/home/cotuex/.local/bin/uv run uvicorn app.controller_api:app --port 3000
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

Fig. A.1: Controller API service definition

ini

controller-software.service

```ini
[Unit]
Description=Controller software
Requires=controller-api.service
After=controller-api.service

[Service]
Type=simple
WorkingDirectory=/home/cotuex/pilock/controller
ExecStart=/home/cotuex/.local/bin/uv run main.py
Restart=on-failure

User=cotuex
Group=cotuex

[Install]
WantedBy=multi-user.target
```

Fig. A.2: Controller software service definition

ini

manager-web.service

```ini
[Unit]
Description=Manager web
After=network.target

[Service]
Type=simple
WorkingDirectory=/home/cotuex/pilock/manager-web-server
ExecStart=/home/cotuex/.local/bin/uv run uvicorn app.main:app --host
0.0.0.0 --port 80
Restart=on-failure

User=cotuex
Group=cotuex

[Install]
WantedBy=multi-user.target
```

Fig. A.3: Manager Web server service definition

Appendix B - Management Web Server UI

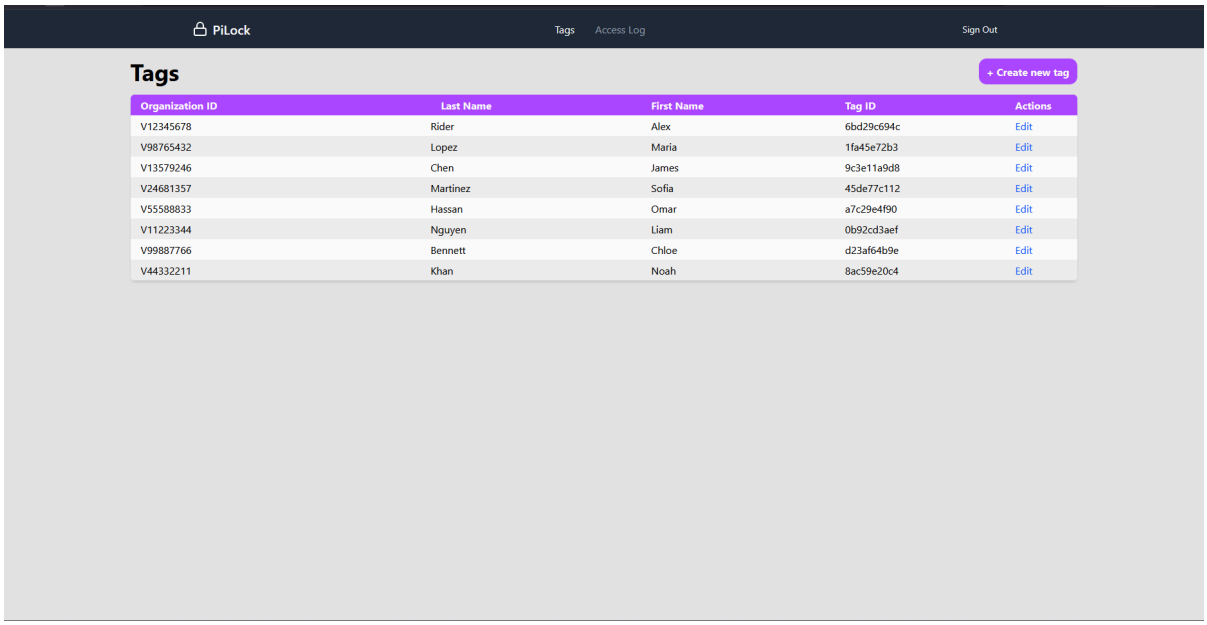


Fig. B.1: Tags - List

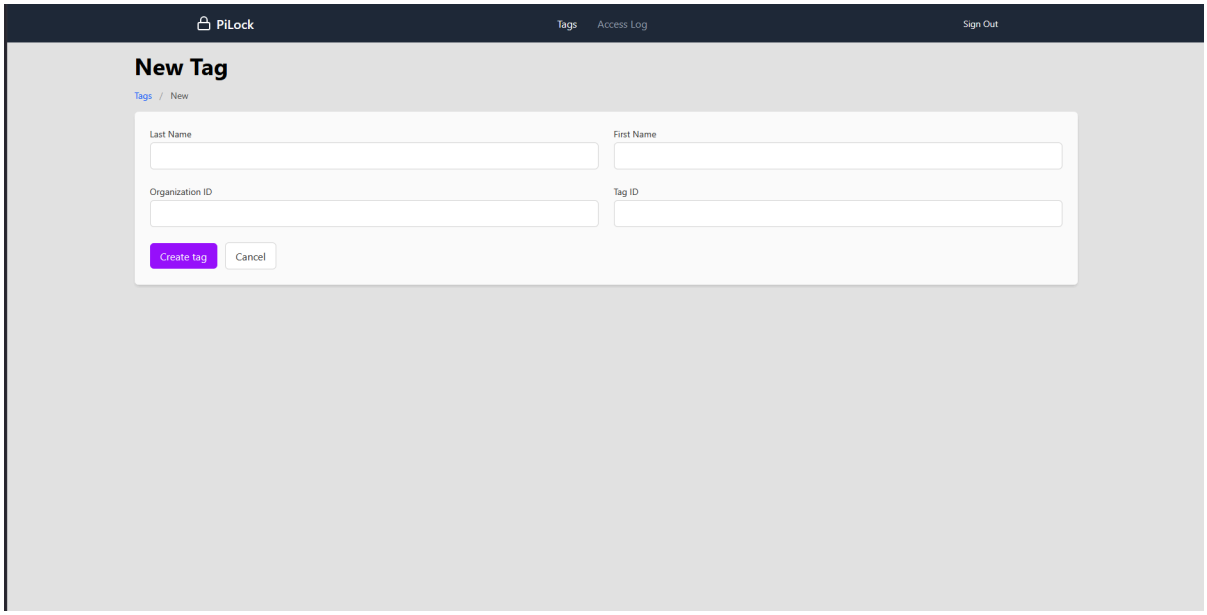


Fig. B.2: Tags - Create

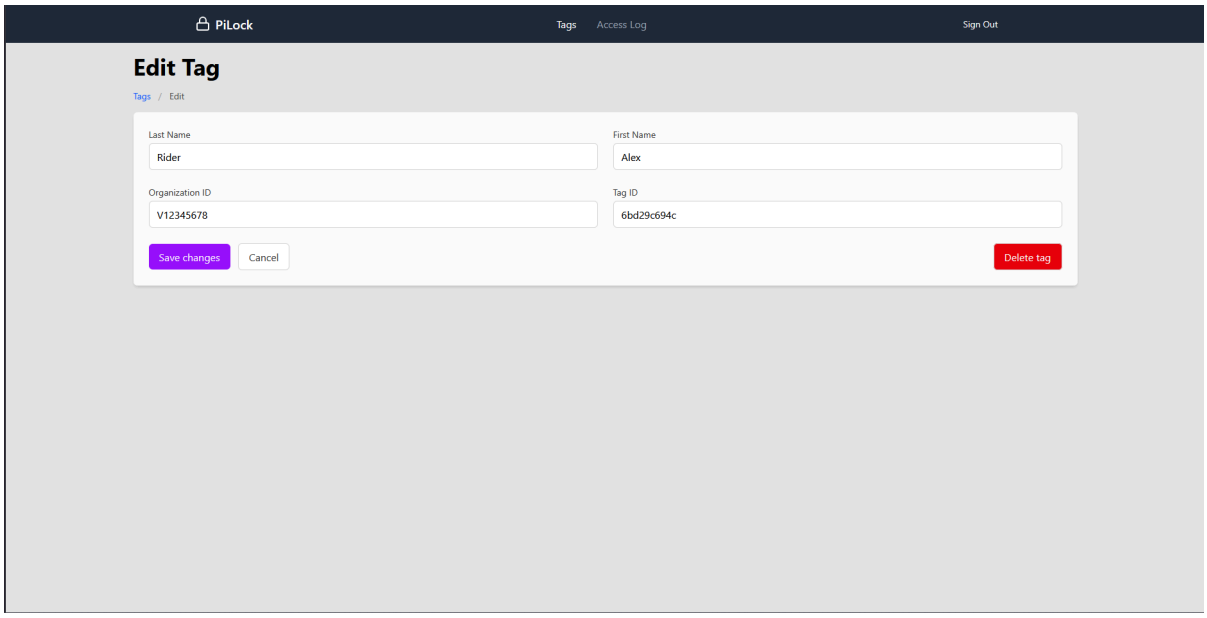
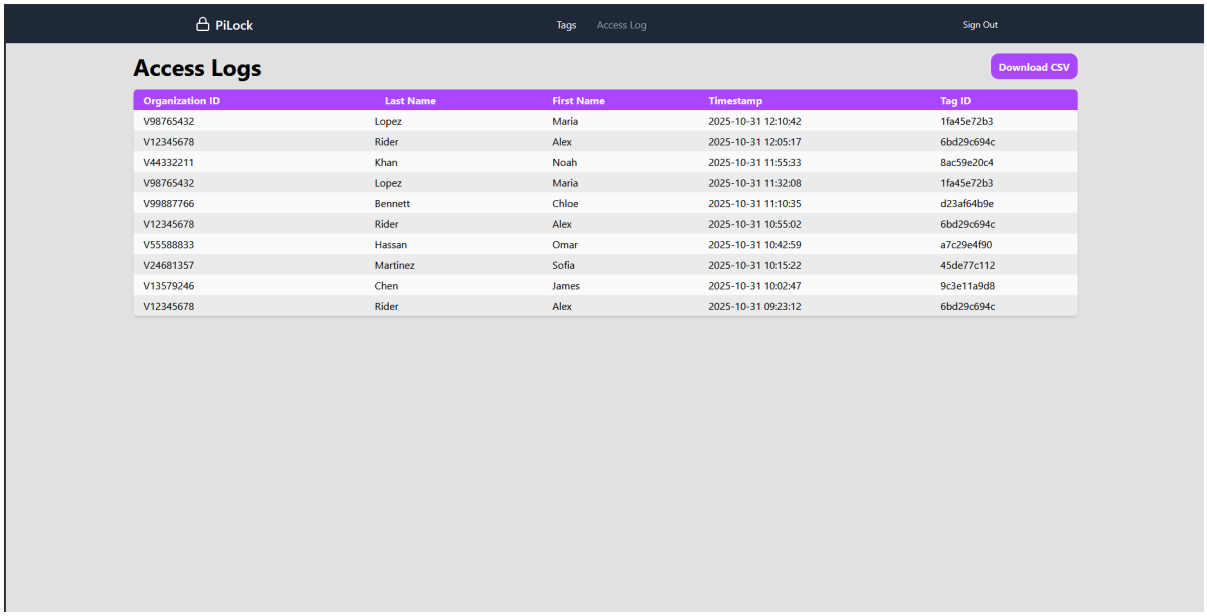


Fig. B.3: Tags - Edit



Organization ID	Last Name	First Name	Timestamp	Tag ID
V98765432	Lopez	Maria	2025-10-31 12:10:42	1fa45e72b3
V12345678	Rider	Alex	2025-10-31 12:05:17	6bd29c694c
V44332211	Khan	Noah	2025-10-31 11:55:33	8ac59e20c4
V98765432	Lopez	Maria	2025-10-31 11:32:08	1fa45e72b3
V99887766	Bennett	Chloe	2025-10-31 11:10:35	d23af64b9e
V12345678	Rider	Alex	2025-10-31 10:55:02	6bd29c694c
V55588833	Hassan	Omar	2025-10-31 10:42:59	a7c29e4f90
V24681357	Martinez	Sofia	2025-10-31 10:15:22	45de77c112
V13579246	Chen	James	2025-10-31 10:02:47	9c3e11a9d8
V12345678	Rider	Alex	2025-10-31 09:23:12	6bd29c694c

Fig. B.4: Access Logs - List

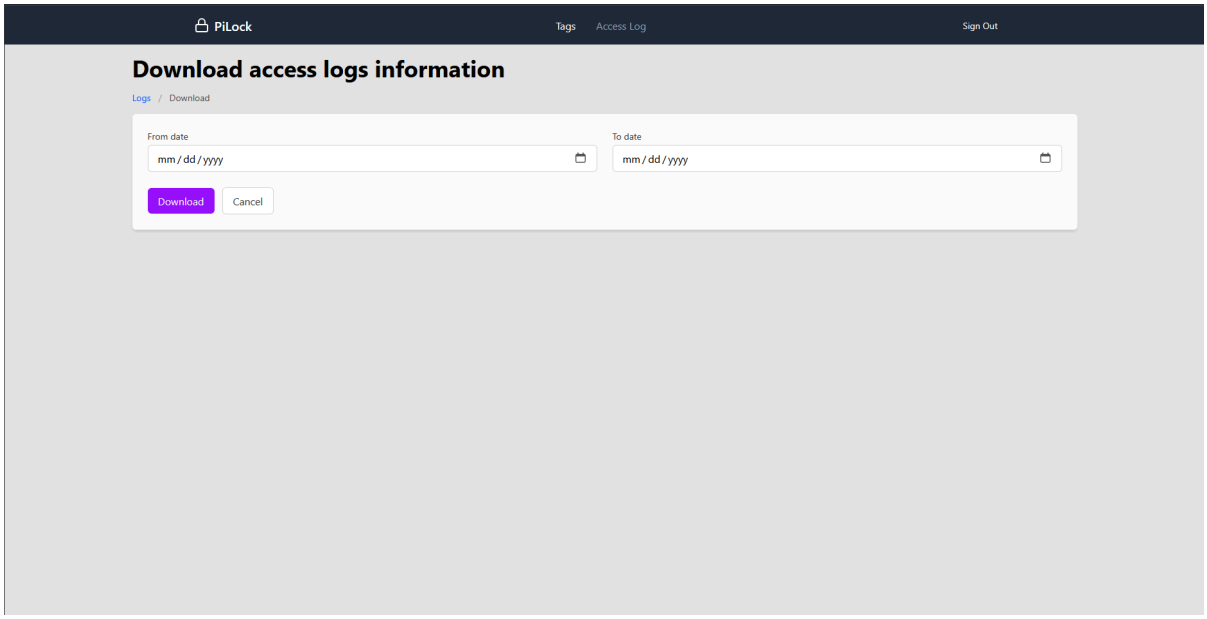


Fig. B.5: Access Logs - Download

Appendix C - Python code

The full code for PiLock system can be found in the following github repo:

```
https://github.com/josuebouchard/pilock
```

Listing C.1: Github repository link for PiLock code

Manager Web Server

Web Interface Server

```py
@tag_router.get("/", name="tags:index-ui")
def tag_index_ui(request: Request, session: SessionDep):
    tags = get_tags(session)
    return templates.TemplateResponse(
        request,
        name="tags/index.jinja",
        context={"tags": tags},
    )
```

Listing C.2: Example endpoint: get all tags

```py
from typing import Any
from sqlmodel import SQLModel, Field
from datetime import UTC, datetime

class PersonInformation(SQLModel):
    org_id: str
    first_name: str
    last_name: str

class _TagBase(PersonInformation):
    tag_uid: str

class Tag(_TagBase, table=True):
    __tablename__: Any = "tags"

    id: int | None = Field(default=None, primary_key=True)

    def person_information(self):
        return PersonInformation.model_construct(**self.model_dump())

class TagEdit(_TagBase):
    pass

class AccessLog(SQLModel, table=True):
    __tablename__: Any = "access_logs"

    id: int | None = Field(default=None, primary_key=True)
    tag_uid: str
    access_was_granted: bool
    timestamp: datetime | None = Field(default_factory=lambda:
datetime.now(UTC))

    org_id: str | None = Field(default=None)
    first_name: str | None = Field(default=None)
    last_name: str | None = Field(default=None)
```

Listing C.3: Database models

```py
def get_tags(session: Session):
    return session.exec(select(Tag)).all()


def get_tag_by_id(id: int, session: Session):
    return session.exec(select(Tag).where(Tag.id == id)).first()

def get_tag_by_tag_uid(tag_uid: str, session: Session):
    tag_uid = tag_uid.strip().upper()
    stmt = select(Tag).where(col(Tag.tag_uid).ilike(tag_uid))
    return session.exec(stmt).first()

def save_tag(tag: Tag, session: Session):
    tracked_tag = session.merge(tag)
    tag.tag_uid = tag.tag_uid.strip().upper()

    session.commit()
    session.refresh(tracked_tag)

    return tracked_tag

def disable_tag(tag_id: int, session: Session):
    tag = session.get(Tag, tag_id)
    session.delete(tag)

    session.commit()
```

Listing C.4: Tag related repository

```py
def get_access_logs(session: Session):
    stmt = select(AccessLog).order_by(col(AccessLog.timestamp).desc())
    return session.exec(stmt).all()


def create_access_log(
    tag_uid: str,
    access_was_granted: bool,
    person_information: PersonInformation | None,
    session: Session,
):
    access_log = AccessLog(tag_uid=tag_uid,
access_was_granted=access_was_granted)

    if person_information is not None:
        _ = access_log.sqlmodel_update(person_information)

    # {"tag_uid": tag_uid, "access_was_granted": access_was_granted}
    session.add(access_log)
    session.commit()
    session.refresh(access_log)

    return access_log
```

Listing C.5: Access log related repository

Controller API

```py
@app.get("/access/check")
def check_access(tag_uid: str, session: SessionDep):
    maybe_tag = repository.get_tag_by_tag_uid(tag_uid, session)
    is_access_granted = maybe_tag is not None
    maybe_person_information = maybe_tag and
maybe_tag.person_information()

    access_log = repository.create_access_log(
        tag_uid=tag_uid,
        access_was_granted=is_access_granted,
        person_information=maybe_person_information,
        session=session,
    )

    return access_log
```

Listing C.6: Controller of the only endpoint exposed by the Controller API

Controller software

```python
def main():
    log.info("system start-up")
    try:
        # Poll the NFC sensor
        while True:
            attempt_tag_read()
            attempt_clean_last_seen()

            time.sleep(0.05)  # Prevent 100% CPU usage

    except KeyboardInterrupt:
        rdr.cleanup()
        log.info("system stop")
```

Listing C.7: Main loop

```python
# Attempts to read a tag
def attempt_tag_read():
    (error, _tag_type) = rdr.request()
    if error:
        # No card is present
        return

    # Try to get the UID
    (error, uid) = rdr.anticoll()
    if error:
        log.error("NFC anticollission error")
        return

    # Convert UID to hex string
    hex_uid = bytearray(uid).hex()

    # Time based debouncing
    now = time.monotonic()
    if (now - last_seen.get(hex_uid, 0)) >= DEBOUNCE_SECONDS:
        # Action:
        on_NFC_tag_detected(hex_uid)
    last_seen[hex_uid] = now
```

Listing C.8: Code to read UID from RC522

```python
# Performs action on NFC detected
def on_NFC_tag_detected(hex_string_uid: str):
    nfc_log = log.bind(nfc_tag_uid=hex_string_uid)

    try:
        req = http_client.get(
            "/access/check",
            params={"tag_uid": hex_string_uid},
            timeout=2.0,  # Slow server cannot block system
        )
    except httpx.RequestError as exc:
        nfc_log.error(
            "HTTP request failed",
            error=str(exc),
        )
        return

    if req.is_error:
        nfc_log.error(
            "HTTP validation request error",
            status_code=req.status_code,
        )
        return

    response = req.json()

    nfc_log = nfc_log.bind(remote_access_log=response)

    if response.get("access_was_granted", False):
        lock.toggle()
        time.sleep(0.5)
        lock.detach()
        nfc_log.info("access granted", lock_state=lock.is_locked)
    else:
        nfc_log.info("access rejected", lock_state=lock.is_locked)
```

Listing C.9: Action on NFC tag detected

```python
# Prevents last_seen dict to grow infinitely
def attempt_clean_last_seen():
    global last_clean
    now = time.monotonic()
    if now - last_clean < CLEAN_PERIOD_SECONDS:
        return

    stale_uids = [
        hex_uid
        for hex_uid, last_seen_timestamp in last_seen.items()
        if now - last_seen_timestamp >= DEBOUNCE_SECONDS
    ]

    for hex_uid in stale_uids:
        del last_seen[hex_uid]

    last_clean = now
```

Listing C.10: Subroutine to prevent time-debouncing dictionary from growing unbounded