TapID: An Expandable Unique Identification System Facilitating Campus Interaction Data Management

Enrique Miguel Jose E. Villa
Contemporary Media and Communications Strand
The Little Farm House Holistic Education & Development Center
Inquiries, Investigations and Immersion
Romar Lintag
Career Advocacy
Glen C. Martir
April 20, 2022

Endorsement

This thesis hereto attached entitled **TapID: An Expandable Unique Identification System Facilitating Campus Interaction Data Management**, prepared and submitted by Enrique Miguel Jose E. Villa in partial fulfillment of the requirements for the General Academic Strand, is hereby accepted.

Henry G. Calilung
School Head

Rudolph Arnold A. Rabago
Senior High School Coordinator

Glen C. Martir
Track Mentor

TapID: An Expandable Unique Identification System Facilitating Campus Interaction Data Management

Enrique Miguel Jose Emano Villa, *Student, The Holistic Education and Development Center*

***Abstract*—Banknotes have been used a way to physically pay all throughout the world. School systems use chits to teach children about buying and saving money, and to try and create a sanitary and school-controlled currency. This leads to a large problem as students share chits, adding another medium for pathogens to spread, and students lose multiple chits which cause the school to produce more waste. This study creates an exapandable technological ecosystem named TapID for school campuses to prevent this problem. Results prove that the system is feasible (first prototype is via a Raspberry Pi Pico and a Python HTTP API), could improve waiting times creating more efficient interactions, and is within a reasonable price for a product.**

***Index Terms*—Money, Identification, JSON Web Tokens, Python, Raspberry Pi Pico, Electronics, Expandable, Microcontrollers, PostgreSQL**

I. Introduction

Banknotes are rectangular-shaped bills that are used as a mode of payment in countries with a currency system-based economy [1]. These banknotes are actively used as a method of physical payment worldwide. Schools allow students to use banknotes for payment in their facilities due to their ease of use and reliability. Most banknotes are manufactured with either a mix of cotton (such as the Philippine bills) or polymer, both of which allow the growth of single or multiple drug-resistant pathogens [2]. The combination of the bills' widespread use [3] and its material make the bill a very efficient medium for bacteria and viruses to spread through.

This study proposes an open-sourced technological ecosystem named TapID to reduce the spread of pathogens by removing the need for paper bills. It utilizes RFID (radio-frequency identification), asymmetrically encrypted JWTs (JSON web tokens), microcontrollers, and an HTTP server (named TapAPI) to create an unclonable and unique identifier for a person, which then could be used for secure canteen transactions. This identifier could be used for more than tracking canteen payments, as it works as a "digital signature". Any interaction that requires another persons signature (and optionally data) will be called an "authenticated interaction", and support for all of these interactions could be implemented by future researchers via the HTTP server's plug-in architecture. The system's expandability will be illustrated in this research by providing 3 extensions that can be used as examples: a library tracker extension, the tracking of in/out times and canteen transactions.

There are two reasons why TapID is scoped to only function within school campuses: innovation and protection. The expandability of the system allows future aspiring computer science researchers to add clean and maintainable code to a beginner-friendly open-sourced system. The second reason — protection — is due to children having a weaker immune system compared to adults [4]. Because of the project's open-sourced nature, any school can implement the technology as long as they are willing to shoulder the installation, maintenance, and costs. All these features are needed to make the system convenient for students and teachers alike by increasing productivity and creating an "open" study where future researchers can add their own features and/or improvements by simply understanding how the system works through the open-sourced features previously mentioned.

II. Review of Related Literature

This literature review separates the transaction systems into two categories: "stationary", meaning data is processed on the module, and "split", which means the system is split in two, one for reading the card data, and another for processing it. It is important to note that the possible system security vulnerabilities/weaknesses are possible problems that hypothesized to arise solely from the system details indicated on their respective papers. If a paper does not provide enough detail (or does not go into detail about a certain aspect of a system), then a security problem may not be observed.

A. Stationary systems

1) Dewanto et al. [5]: Dewanto et al. proposes a design that uses RFID as the sole basis in their security for their transaction system. Radio-frequency identification (RFID) tags are a lightweight and portable way to transmit data invented last 1970 to track large items (luggage, livestock, cars, etc.). More recent iterations of the technology made RFID the ideal way to store small amounts of information in a sticker or card ranging from 1 to 2 kilobytes. Data is transferred using electromagnetism to power the RFID's thin coil allowing it to backscatter (i.e., send back) radio waves in a set frequency to return the data. There are currently

multiple types of RFID cards, each with its own pros and cons [6].

TapID and Dewanto et al.'s [5] study utilizes passive RFID tags in its identification system. Passive RFID tags require the reader's antenna to send the power required for a passive RFID card to backscatter the data stored inside it. In contrast, active or semi-passive tags require batteries stored within them to power them. This makes the passive RFID ideal for the system as TapID's method of identification does not require the long-distance reading of RFIDs, unless needed in a future Ground Module plug-in, which in that case can be implemented by future contributors. It is also disposable and cheaper to manufacture compared to the other types [6].

The card verification is done through a "merchant password" code written on an RFID's storage before usage. The aforementioned password was not expounded on by the researchers of the paper, but the two possible options for this password a string of ASCII letters or an array of 8-bit integers ranging from 0-255, as these are the only data types that are writable on an RFID card. If a detected card does not have a password written on the sector it is meant to be on, or if a card's password does not match the merchant password (possibly due to an individual trying to penetrate the system), then the transaction is declined.

To deploy this method, each RFID in the system must read by a RFID reader and writer and loaded with the appropriate password before use. If an organization requires 1000 cards, a person or group must manually tap each card on the reader to write the master password on each cards' password sector. Unfortunately, there is no other way to easily write on RFID cards in bulk which means this very demanding task is unavoidable even for TapID as every individual's JSON Web Token must be written on their card. Writing a JSON Web Token (the identifier TapID uses to verify a person's identity) takes an estimate of 3 seconds, so card preparation must be done at an estimate of at least 2-3 weeks before deployment to allow for errors and proper setup.

Dewanto et al.'s [5] wiring diagram (Fig. 1) is very straightforward. The RFID reader (MFRC522) and 16x2 LCD will be present in TapID's ground module because they cost relatively cheap, are reliable, and Arduino libraries (abstracted code wrapped into functions to interface with the electronic modules) have already been written for them in Arduino's language (C++) reducing the amount of work needed to implement a system with Arduino utilizing them. With small projects like Dewanto et al.'s [5] and TapID, open-sourced programmable microcontrollers are the best fit for prototyping and executing the system. A custom-designed PCB would be possible as well, but given the time restrictions of TapID it may be unfeasible due to PCB manufacturing and shipping times.

The system's software flowchart begins when a card is detected by the sensor. It first validates if the "merchant password" is present and correct, then it checks if their balance is greater than or equal to the amount requested, and if the user has enough balance, it subtracts the amount from

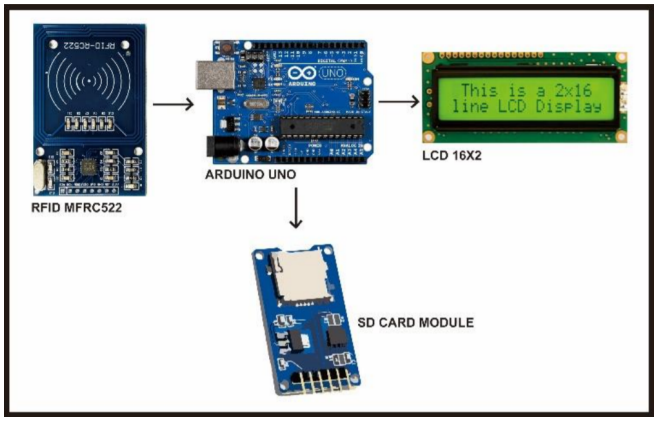


Figure 1: Dewanto et al's abstracted wiring diagram.

the user's balance stored on the card and logs the transaction on an excel file present in the SD card module [5].

The system stores a user's balance directly on the card. This makes it difficult to recover a card's balance once it is broken unless the logs are observed, which is difficult due to another problem related to the SD card: if an individual were to request the data in the SD card, the system must not be active to safely remove the SD card or else the system would crash upon trying to log something on an empty card bay. If the error is instead properly caught and isolated in the software, the logs would be incomplete, as there is nothing to write on.

Another problem related to that is if the SD card were to fail at ˜10.6 x 10^7 writes [7] in software where errors are handled silently. The system observers would not be aware of the card failure because the SD card module cannot differentiate a broken/corrupted card inserted in the module from no SD card inserted at all. A completely broken SD card would throw the same error as having no card inserted. These two reasons are why SD card modules are not implemented in the TapID ground module; they add another layer of complexity in accessing and reliably storing data.

2) Attar et al. [8]: Attar et al.'s system (flowchart seen in Figure 2) is somewhat similar to TapID in the data-storage aspect. It uses a Python script to interact with an SQL database in a computer separate from the Arduino. The paper is very brief and did not go into technical details about the system, but one can speculate that Attar et al. [8] utilized Arduino's serial communication to interface with Python and order the script to query a database with the information given through serial, as it is an easy method to communicate a Python script with an Arduino microcontroller.

Attar et al. [8] utilized an SQL database to store the data required for user authentication. Databases are data structures that can store organized data that can be used dynamically (i.e., on runtime) by computer programs [9]. Data is organized into different tables, each with different fields that require a user-defined type of information (e.g., a letter, a number, pure binary). A database can be queried by using the Structured Query Language (SQL). Its syntax

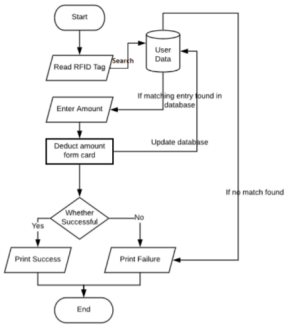


Figure 2: Attar, et al.'s [8] RFID-Based Universal Transaction System Flowchart.

follows a command-like format with clauses, expressions, and predicates (see Figure 3).

They did not go into details about what exactly they are storing in the database, but their abstract mentions that "The users of the system will have to be pre-registered and all the user related information will be saved in a MySQL database." This hints that they most probably store an individual's card UID as the database's primary key alongside their balance, allowing them to query the system easily based on a card.

```
SELECT * FROM canteen_chits
WHERE uid = '4B 69 2D 0C';
```

Figure 3: An example of an SQL command used to query the TapID database.

This system's hypothesized security conclusion cannot be given as the researchers were not specific in terms of what they read to authenticate a user. If they read a "passphrase" stored on the card, then it is more unsecure compared to when it reads a cards UID as a card UID is not changeable on certain card models, making it unique to one person. An RFID card's UID has 4,228,250,625 (255^4) possible combinations, which is certainly more than enough for a small organization. But data-storage wise, this system's centralized database for user balance/info storage is ideal, as data of this type is best stored on a relational database [10]. A database on each ground module would be very computationally expensive and complex (i.e., difficult to keep in sync) once the modules reach a high amount [11]. The only disadvantage in the system is that each module requires to be connected to a computer running a Python script. Attar et al.'s [8] also did not mention if the Python script can handle multiple modules.

B. Split systems

1) Caya et al. [12]: Caya et al. [12] system presents a system that stores an individual's balance directly on the card — which was previously discussed to be a problem — but the system stores the transaction logs on an SQL database located in a separate computer and connects to it using an Ethernet module (see Figure 4). Once a card is broken, the database can easily be queried without blocking other processes. When compared to the SD card solution (which would require the entire module to halt before data analysis), it is much better. This architecture is what TapID is leaning towards, as it is efficient and safe for data storage.

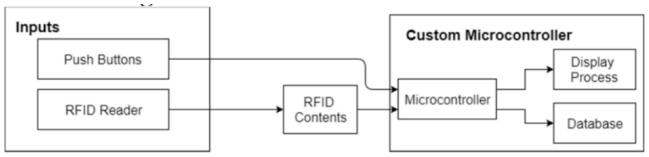


Figure 4: Caya et al.'s [12] general system diagram.

Caya et al. [12] uses the Arduino UNO powered by the ATmega328P while using an Ethernet module to connect to the database. TapID will also utilize the internet to send interaction data to the API, but wirelessly through Wi-Fi using an ESP-01 module communicating with the Raspberry Pi via UART. Although Wi-Fi has a higher packet loss due to interference in the air as compared to Ethernet cables [13], it is still reliable enough as long as the entire network is wired efficiently. This can be achieved by either tapping into a schools current Wi-Fi network or by preferably creating a secondary network to give the TapID ground modules all the bandwidth to communicate with the API even during student Wi-Fi usage peak times.

2) Siregar et al. [14]: Siregar et al. [14] proposes using a different system for wirelessly communicating with the external SQL database by using the nRF24L01 RF transceiver module. This is a creative use case of the module, and the module is robust data-transfer wise as it can support up to 800m of communication (with a clear line of sight; [15]). Using RF to send data is not inherently bad, and definitely has a use case in multiple scenarios, but in a school campus (such as this study is scoped in), Wi-Fi presents itself as a better alternative.

Siregar et al. [14] proposes a "master" and "slave" design (Figure 5) using an Arduino Nano based board (STM32F103C8T6) as their microcontroller for both. The master board communicates with the SQL database using an Ethernet module, and the slaves communicate with the master. Siregar et al. [14] also created a front-end website to facilitate transactions for cashiers, which is not needed in the first version of TapID, but can be added by a future researcher.

This appears to be a robust system, but one limitation not disclaimed in the paper would be the nRF2L01's trademarked MultiCeiver feature. When multiple slave RF nodes (PTX) connect to a single master (PRX), according to the nRF2L01's data-sheet, only "Up to six nRF24L01+s configured as PTX can communicate with one nRF24L01+ configured as PRX." [15]. This would mean that once the total amount of 6 modules connected to the master module have been reached, another master node will have to be

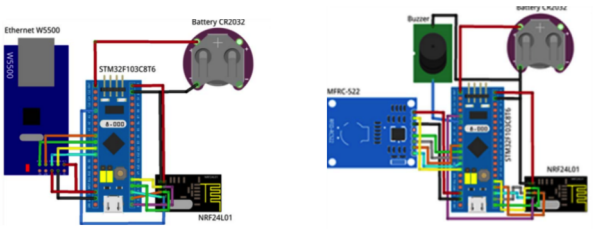


Figure 5: Siregar et al.'s [14] system cabling diagrams. Master (left) and slave (right).

created to communicate with the system. This at a small scale would be efficient, but TapID is designed to be modular which is why the Ground Modules utilize Wi-Fi, as it handles as many devices as the network's bandwidth can handle, and TapID at minimum only sends an approximate 400-byte request to TapAPI (see Figure 6).

An Application Programming Interface (API) is defined as a "set of commands, functions, protocols, and objects that programmers can use to create software or interact with an external system" [16]. In an HTTP web-server context, it allows users to request data from a server, with the server responding with the data or an error code. TapID utilizes a web server architecture called a REST (Representational State Transfer) API. This type of API modifies data saved in a database when requested by users [17]. This concept of a REST API will be the basis of the second half of TapID's system, aptly named TapAPI, which will be the heart of TapID's data collection and processing.

```
POST /event /HTTP/1.0
Host: 127.0.0.1
Content-Type: application/json
Content-Length: 43

{
    "jwt": "eyJh...t04Li6GdeUM",
    "event_name": "canteen_event",
    "event_data": {
        "action": "subtract",
        "bal": 50
    }
}
```

Figure 6: An average TapID HTTP POST request.

The ESP-01 used in the TapID ground module has a maximum of a 115200bps (bits per second) upload speed, and the lowest local area network (LAN) network speed is 100mbps (higher could be achieved with specialized equipment [18]) which is more than enough for handling small requests. Dividing the maximum upload speed by the average request size, it would take the module 288 requests in a single second to overload the Wi-Fi module (see Figure 7).

The theoretical maximum of 288 API requests in a single second is realistically impossible to happen unless a software glitch were to occur (such as a never-ending while loop), as

$$\frac{115200 \text{ bytes/second}}{400 \text{ bytes/request}} =\sim 288 \text{ requests/second}$$

Figure 7: Maximum amount of TapID requests per second.

it takes an average of 3 seconds for the TapID v1 Ground Module to respond. Once the limit has been reached, a small delay will only be noticed when using the Ground Modules. It will definitely bring an arbitrary amount of system load to the API depending on the events asked.

III. Methodology

A preliminary survey was conducted to determine which of the three built-in plug-ins are of top priority in terms of efficiency, speed, and reliability. The survey was given via convenience sampling to all grade 6-11 students in The Holistic Education & Development Center (HEDCen) who have bought in their school campus' canteen at least three times, and/or have borrowed equipment in their library or science laboratory and have consented to taking the survey. The survey was analyzed quantitatively, as a real average purchase duration could not be possible given the current online schooling conditions. There is an advantage to these Likert-scale based questions to determine duration, as students answer what they feel and everyone's perception of time could be different. Descriptive statistics was used to compare the people who wished for the system to be faster and those who did not wish for it to be faster. The full list of survey questions can be viewed at the link attached in Appendix A.

Multiple software were used in the initial prototyping and development of the software and hardware for the technological TapID ecosystem. All programs were ran on MacOS Monterey 12.1. Autodesk Fusion 360 was used in creating the physical 3D prints for the Ground Module, while Ultimaker Cura was used to slice the models for printing. The .3mf (3D model) files were printed on the Creality Ender 3 v2 3D printer using gray 1.75mm eSUN PLA+ filament. Python 3.9.9 was the main language used throughout the project, MicroPython being the subset of Python used to program the Raspberry Pi Pico microcontroller. Python was used in it's microcontroller variant as well to prevent confusion/delays in learning a new language for future learners who wish to contribute to the ecosystem. The JetBrains PyCharm IDE was the main tool used for development of the Python Flask HTTP API and the code for microcontroller, while the JetBrains DataGrip IDE was used for tweaking the PostgreSQL database. Multiple Python and MicroPython libraries have been used in the system to increase development speed and code readability. The IDEs used are not required, but was used because of the researcher's familiarity. All the code has been incrementally committed to the GitHub repository during development.

IV. Results and Discussion

A. Preliminary survey

The preliminary survey was created using Google Forms and given out through convenience sampling. The raw data can be seen in Appendix I. The survey aided in determining the order of development for the plug-ins, which plug-ins could lose some readability in order to improve speed, and what to tailor the Tap ecosystem to handle for the next future projects. Figure 8 determines what feature of TapID should development go in the most. 12 students were able to take the survey. The results show that the users prioritize ease of use, then reliability, and then durability, and they least prioritize speed.

The results for the yes/no answers in the survey can be seen in Table I. Note that the students were segregated before being asked questions about the library and the canteen system. Only students who have bought in the canteen more than 3 times are allowed to answer the canteen questions to ensure that their estimates are moderately accurate. For the library/lab equipment borrowing, students are required to have borrowed at least once.

The results show that half of the students have shared chits with their schoolmate and that there is a sanitary problem as majority of students report seeing other students use pens that other people have touched and have seen the canteen circulating chits that other schoolmates have recently touched. The students seem to be neutral towards the multiple features of the canteen (see Figure 9).

The option for a minimum top-up amount was explored to account for royalties, electricity, and maintenance, but 8 of the 12 students wish to not have a minimum top-up amount. Students seemed to like the concept of tapping your ID on a sensor to record attendance the most, and seeing a live map of where all your teachers are the least (see Figure 11).

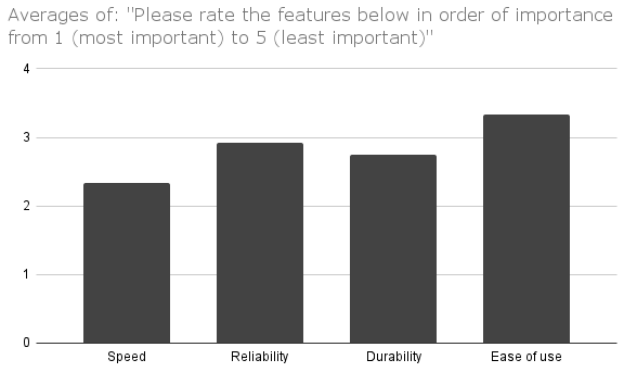


Figure 8: Averaged results for ”Please rate the features below in order of importance from 1 (most important) to 5 (least important)” in context of the TapID ecosystem.

B. TapAPI

All the code (including the .tex source for this paper) for the entire TapID ecosystem is available in the repository

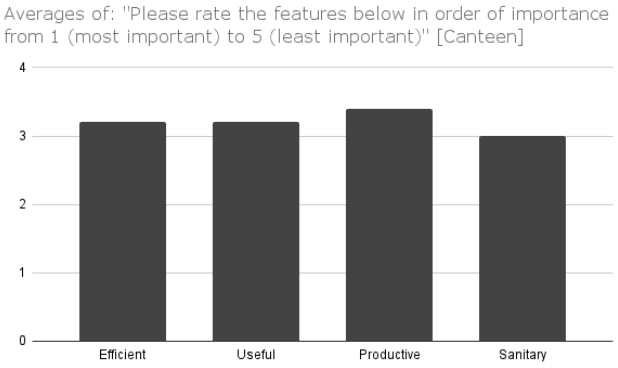


Figure 9: Averaged results for ”Please rate the features below in order of importance from 1 (most important) to 5 (least important)” in context of the canteen system.

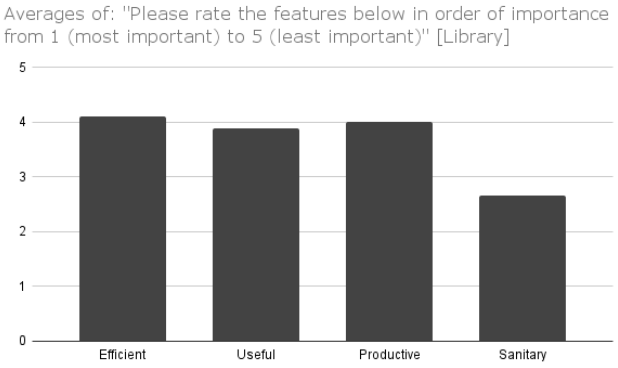


Figure 10: Averaged results for ”Please rate the features below in order of importance from 1 (most important) to 5 (least important)” in context of the library system.

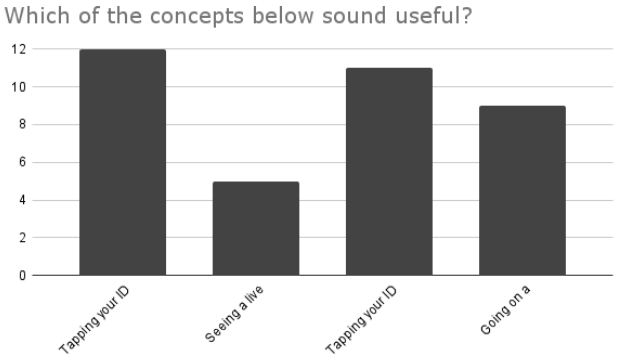


Figure 11: The amount of students who think the following concepts are useful. From left to right: ”Tapping your ID on a sensor to record attendance”, ”Seeing a live map of where all your teachers are”, ”Tapping your ID on a sensor to surrender/un-surrender your gadgets from the Admin office”, and ”Going on a website to see what books you have borrowed and when they have to be returned”

Question	Yes	No
General questions		
Would you use a card-based canteen payment system?	7	5
Have you ever shared chits with a classmate?	6	6
Have you seen the canteen circulating chits that other schoolmates have recently touched?	7	5
Have you ever seen people use pens that other people have used to write their signatures?	8	4
Library system (9 people)		
Would you wish for this system to be faster?	8	1
Canteeen system (5 people)		
Would you wish for this system to be faster?	5	0

Table I: Survey results for the binary questions.

linked in Appendix B. TapAPI was the priority and was made first in development to allow for easy iteration for the Ground Module prototypes as the Ground Module heavily relies on the API to be able to perform its sole function. The code for the server can be seen in the repository's /tapapi/ directory. Table III lists down the multiple non-standard (i.e., did not come with the default Python installation) Python libraries used in the API and their function. Figure 12 displays the general system diagram. Note that not all the libraries are seen in main.py, as some are located in other Python files used in abstracted functions to keep the look of the main file clean and readable while maintaining function. The TapAPI also supports command-line arguments when running main.py, some are required to run the server (see Table II). These arguments are passed to the `run()` function of all plug-ins in case it is needed (see Table IV).

Long name	Short name	Required?	Use
`--level`	`-l`	No	Sets logging level.
`--port`	`-p`	No	Sets the network port that TapAPI will run in.
`--db-user`	`-user`	Yes	The PostgreSQL database username.
`--db-pass`	`-pass`	Yes	The PostgreSQL database password.

Table II: Command-line arguments and their usage

1) JSON Web Tokens and Authentication: JSON Web Tokens are a secure way to transfer information as JavaScript Object Notation (JSON) object. It appears as a long string of characters separated in 3 parts by periods [19]. It contains a header (contains information about the encryption algorithm and algorithm type), body (contains the information being sent itself), and signature (acts as a checksum of both the header and body). The token is formerly a JSON object, but encrypted using any encryption algorithm.

The encryption algorithm used in TapAPI's tokens is the Rivest-Shamir-Adleman (RSA) asymmetric encryption algorithm. It uses two keys (mathematically labeled p and q) which are two very large prime numbers. In RSA, the public and private keys are commutative, meaning one can can decrypt a message encrypted by the other [20]. The body of the JSON Web Token holds the students full name, grade, a password and a UUID based on the RFC4122 standard. Storing this information in the JWT makes sure that there is

TapAPI v1 main system flow diagram

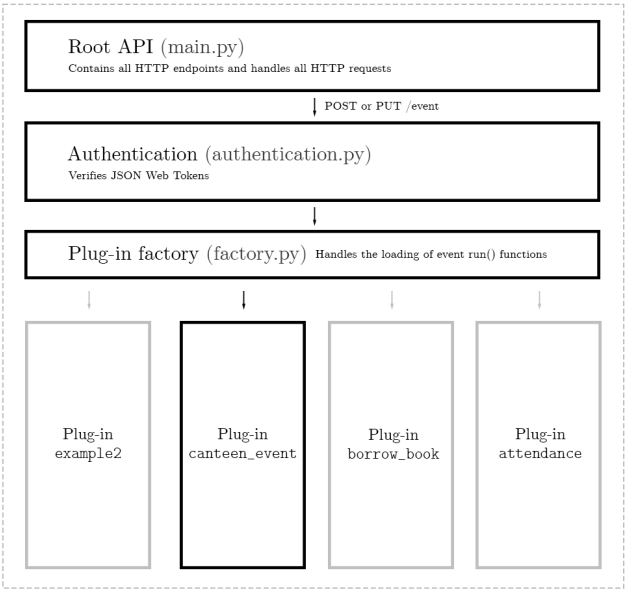


Figure 12: TapAPI general authentication to plug-in system diagram.

Name	Author(s)	Use
PyJWT	Jose Padilla	To be able to easily decrypt/encrypt the JSON web tokens stored inside the RFID card.
Flask	Armin Ronacher	Used to create the server's REST API and website.
Psycopg2	Federico Di Gregorio	Used to interface with the PostgreSQL database.
Cryptography	The Python Cryptographic Authority	To create the RSA-256 key pairs.
Pipenv	Pipenv maintainer team	For easy package organization and installation.

Table III: The list of libraries used in TapAPI and their respective functions.

no table in the database filled with every students' full name and grade, which is sensitive information.

The format is enforced (minus the UUID, as it is not required in this version of the API) in the root API, and any decrypted JWTs with an invalid format or with an incorrect password will immediately be ignored and fail authentication even if decrypted with the card UID's assigned public key. The full code can be viewed in the JWT creation chapter of the TapID wiki (see Appendix C) and in Appendix D. The body is then encrypted with a private key, and the public key is stored in the public_keys table present in the TapID database along with it's designated 4-byte UID.

The JWT is encrypted using an RSA public exponent (e) of 65537 ($2^{16}+1$ or `0x10001`) as it is the most commonly used exponent, and the choice of exponent does not usually affect the encryption's security unless a small e is chosen, such as 3 [21]. Technically, any prime number greater than 2 can work as a public exponent, but Fermat primes are preferred because they are easily findable prime due to it being of form $2^{2^n}+1$. The key size is then set to 2048. TapID private and public keys are stored in PEM (Privacy Enhanced Mail) encoding in PKCS8 format for the private and PKCS1 and public key. All these settings were chosen because it is the standard in the industry but further research is required on its security in our implementation.

```
-----BEGIN RSA PUBLIC KEY-----
MIIBCgKCAQEA6LK2ugUXOHPsXnqk6+ZRFb7cyBzQacjzT4CQf2TmjZmQ8AuunEV3
3bKTLk4GgxCnU9VbgzrodyqUT59b/XUqgQnaHLuALlJMiHFhfwygej8iLnfRZEg3
nW/TvpjhqECf/fitPXBndzrb2KzoqUH0Nve/i6/9CTi53v2iRdwj5VQr9IYBV2Tk
cjytF1b9XWOKV1Vcr2V4FynlZzU+0cCrH4aGY4NuNqMxP7nUBLaCaXeJoUa7PNM1
DQQBoq2ya394H0HaxQlFDX5IOEUa7wwXcHGD4t7clavwOwNLxi6r3rupU6YFAZ3I
+jWgNi99hijxZ+H70aguMx6JZ3wbubk5fQIDAQAB
-----END RSA PUBLIC KEY-----
```

Figure 13: An example of a TapID public key in PEM encoding.

This process could easily be automated in deployment to create a JSON web token per student by simply iterating through a serialized array (such as an excel .csv) through the Python pandas library containing student name and grade information. It is important to note that in the authentication.py function, `get_public_key_from_uid()`, there is an option to save the private key in a test database. This could be useful for testing as creation is automatically done, but the parameter must be set to False in deployment. This would essentially cause the API to think any UID has an associated public key. The pseudocode for the authentication can be seen in Figure 14.

2) TapID database schema: The TapID database uses PostgreSQL (v14.1) as it comes with features that future developers could utilize in their application of the Tap ecosystem. It was created using the default user. The commands listed in Figure 15 describe the multiple tables used in the current release of TapID.

C. Plug-ins and expandability

The API utilizes a plug-in architecture for the TapID system's unique expandability trait using Python's `importlib` library by dynamically importing a Python file. The source code for this sub-section of the API can be viewed in the tapapi/utils/factory.py. The plug-ins are read from the API's /plugins/ directory, but only if specified in the proper format in the config.json file (see Figure 16). This feature allows implementations to disable plug-ins they do not need. Note that plug-ins are called "events" in the source code. Each plug-in is a Python file that must contain a variable named event_name set to the event's name (this is how the factory determines which plug-in to call once a Ground Module requests it), and a function named `run()` that accepts four arguments listed in Table IV (in Python type hinting format, i.e., `name: data type`). These arguments are data structures or classes that allow the plug-in to access useful data that are passed in when an event is triggered.

```
public_key = get_public_key_from_uid(
    uid_from_request)
check for error:
    decrypted_token = decrypt_jwt(
        jwt_from_request, public_key)
if error:
    not authenticated
else:
    send jwt data to plug-ins
```

Figure 14: Pseudocode of the entire authentication process

The `run()` function is essentially the heart of the plug-in, and will run once the respective event name is requested by a Ground Module in it's request body. The plug-in could then use this data freely to to analyze/modify. The plug-in, once done processing data, must then strictly return an instantiated `PluginResponse` data wrapper for the root API to parse, and then return. The reason for this strict return is to prevent plug-ins from returning anything other than an HTTP response code and a JSON payload. This would prevent any accidental mistakes from happening, in turn reducing one point of failure for a crash and increasing the server's reliability.

The returned `PluginResponse` class will then be parsed by the root API, and returned to the Ground Module as an HTTP response for the Ground Module. The plugins are provided an extra /plugins_utils to encourage clean code, so plugins can split their functions into multiple files. Note that when importing packages, import from the root API directory (instead of `from plugin_utils import`, use `from plugins.plugins_utils import`) as the plug-ins are always ran from main.py.

D. Built-in plug-ins

This section describes the built-in plug-ins that handles the canteen transaction, library borrow, and attendance tracking events.

1) Canteen transaction: The canteen transaction plug-in located in /plugins/canteen.py utilizes the canteen_chits table in the TapID database (see Figure 15). A card's balance is stored alongside it's UID. The database automatically

```
create table public_keys (
    uid varchar(11) primary key,
    public_key text unique
);

-- experimental, only for testing
--  do not store all key pairs in one database
--  defeats the security aspect of TapID
create table test_key_pairs (
    uid varchar(11) primary key,
    private_key text,
    public_key text
);

create table canteen_chits (
    transaction_number serial primary key,
    uid varchar(11),
    balance integer
);

create table canteen_transactions (
    transaction_id serial primary key,
    timestamp timestamp,
    uid varchar(11),
    action integer,
    new_bal integer,
    amount integer
);

create table metrics (
    metric_id serial primary key,
    event_name text,
    metric_data json,
    timestamp timestamp
);

create table attendance (
    id serial primary key,
    name text,
    uid varchar(11),
    date timestamp
);

create table book_ids (
    book_id serial primary key,
    book_name text
);

create table books_borrowed (
    borrow_id serial primary key,
    book_id int references book_ids(book_id)
        not null,
    due_on date not null
);
```

Figure 15: SQL commands for creating the TapID database.

generates a transaction ID (which uses it as a primary key) and uses it to keep track of all cards. If a UID does not have an entry in the database, it automatically creates one with a balance of 0. An example HTTP POST request for the Ground Module event can be seen in Figure 17.

A TapID canteen Ground Module can switch between addition and subtraction mode by pressing the A button on the keypad, but this is key programmable within the Ground Module source code. With subtraction mode, the server will process and check if the balance is less than 0 after the operation, which indicates that the balance is less

```
{
  "plugins": [
    "plugins.example",
    "plugins.canteen",
    "plugins.books_library"
  ],
  "version": 1.1
}
```

Figure 16: TapAPI's config.json located in the project's root directory.

Argument Name	Use
`jwt_decoded: dict`	The decoded JSON web token in a Python dictionary format, has information such as the student's name, grade, and uuid.
`payload_data: TapAPIRequestPayload`	A data-wrapper class containing information in the request, such as the `event_data`, the encrypted JWT and the RFID card's UID.
`args`	Contains all `argparse` arguments (i.e., command-line arguments entered when main.py was ran.)
`conn`	A `psycopg2` connection object, allowing the plug-in to access the TapID database and tables within.

Table IV: The list of arguments required to be present in the `run()` function.

than the needed amount, which it will then return an HTTP 403 FORBIDDEN error code. The Ground Module is then programmed to prompt the user with an "Insufficient funds." message on the LCD (see Figure 19).

On a successful balance operation, the server returns an HTTP 200 OK message. Every transaction along with its date of payment is saved to the canteen_transactions table in the TapID database for logging, allowing people to possibly study the transactions. The full code for the canteen transaction event can be viewed in Appendix E and in the GitHub repository. The Ground Module displays a screen that can be seen in Figure 18 on a successful operation. It displays the users new balance, along with a success message.

The maximum balance a user can have is the 4 byte signed number limit, which is 2,147,483,647 (2^{32} but cut in half because of negative numbers). In case a balance more than that number is needed, the canteen_chits table can be modified by using the `alter table` command to use `bigint` (a

```
POST /event /HTTP/1.0
Host: 127.0.0.1
Content-Type: application/json
Content-Length: 210

{
    'jwt': 'eyJ0eXAiO...8QWbA',
    'uid': '4B 69 2D 0C',
    'event_name': 'canteen_event',
    'event_data': {
        'action': 'subtract',
        'bal': 50
    }
}
```

Figure 17: Example HTTP POST request for the canteen transaction event.

type that holds a maximum of 9223372036854775807, or 2^{64}). As of now, there is no currency conversion system, but that can be implemented in future implementations as there are exchange rate APIs that can be used (such as exchangeratesapi.io).

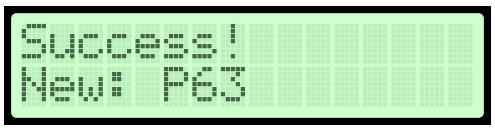


Figure 18: Successful transaction LCD screen.

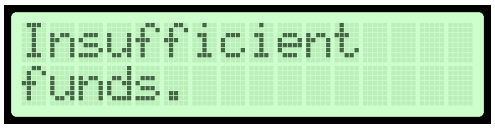


Figure 19: Insufficient balance transaction LCD screen.

2) Library book borrowing: The library book borrowing plug-in showcases TapAPI's ability to integrate with other web services and expandability, in this plug-ins case, the Google Sheets API and an HTML front-end. The source code for this plug-in is viewable in Appendix F and in the GitHub repository. It receives a POST request with event_data containing information about the book that will be borrowed (see Figure 20. The due_on key in the event_data object of the request will tell the API how many days from now the book is due. The book_id solution to entering a book is recommended to be replaced with something easier to input. Each book is assigned an ID, and that ID is input into the Ground Module for it to tell the API which book the user is borrowing. The API then connects to the Google Sheets API and updates a spreadsheet while simultaneously updating it's own local database with the borrow.

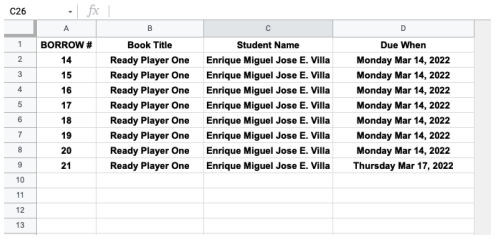
During the initial set-up of the server, a user is prompted with a Google sign-in link to connect their Google Account to the TapID service. This account must have access to the spreadsheet that will log the events. The spreadsheet must also be formatted with 4 columns: BORROW #, Book Title, Student Name, and Due When (see Figure 21). The plug-in's Google Sheet integration utilizes the `sheet.values().append()` function, which means it appends data on empty cells that are under rows that are already filled. The cells in this case are in the A:D range in the spreadsheet.

The plug-in then returns to the Ground Module the same information sent to Google, which the Ground Module uses to display the book's name on the LCD (see Figure 22). It is important to note that the ID inserted into the books_borrowed is assigned as a foreign key, which means it must reference a book in the book_ids table. Trying to insert an ID not assigned to a book will result in an SQL error.

```
POST /event /HTTP/1.0
Host: 127.0.0.1
Content-Type: application/json
Content-Length: 198

{
    'jwt': 'eyJ0eXAiO...8QWbA',
    'uid': '4B 69 2D 0C',
    'event_name': 'borrow_book',
    'event_data': {
        'book_id': 1,
        'due_on': 7
    }
}
```

Figure 20: Example HTTP POST request for the library book borrow event.



	A	B	C	D
1	BORROW #	Book Title	Student Name	Due When
2	14	Ready Player One	Enrique Miguel Jose E. Villa	Monday Mar 14, 2022
3	15	Ready Player One	Enrique Miguel Jose E. Villa	Monday Mar 14, 2022
4	16	Ready Player One	Enrique Miguel Jose E. Villa	Monday Mar 14, 2022
5	17	Ready Player One	Enrique Miguel Jose E. Villa	Monday Mar 14, 2022
6	18	Ready Player One	Enrique Miguel Jose E. Villa	Monday Mar 14, 2022
7	19	Ready Player One	Enrique Miguel Jose E. Villa	Monday Mar 14, 2022
8	20	Ready Player One	Enrique Miguel Jose E. Villa	Monday Mar 14, 2022
9	21	Ready Player One	Enrique Miguel Jose E. Villa	Thursday Mar 17, 2022
10				
11				
12				
13				

Figure 21: An image of the Google Sheet that will log the library book borrows.

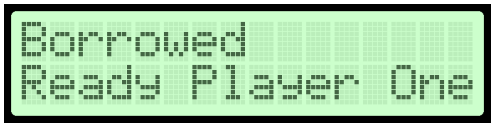


Figure 22: Library borrow book event LCD success message.

The front-end of the plug-in showcases the ability of TapID to work outside of the Tap Ground Modules. When visiting the /library endpoint of the server in a browser, it returns a website where users can enter their card's UID and see what books they have borrowed, and which ones are overdue (see Figure 23). The website utilizes Python Flask's Jinja HTML templating functionality to allow the HTML to dynamically change on different website entries. The website

uses Bootstrap CSS to easily create the aesthetically pleasing UI without any further coding. The table displayed when a UID is entered displays books in red when it past it's due date (see Figure 24).

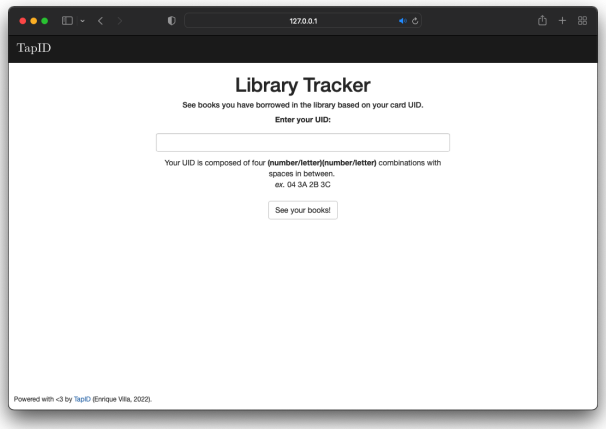


Figure 23: The TapAPI's library website homepage to display borrowed books on desktop and mobile.

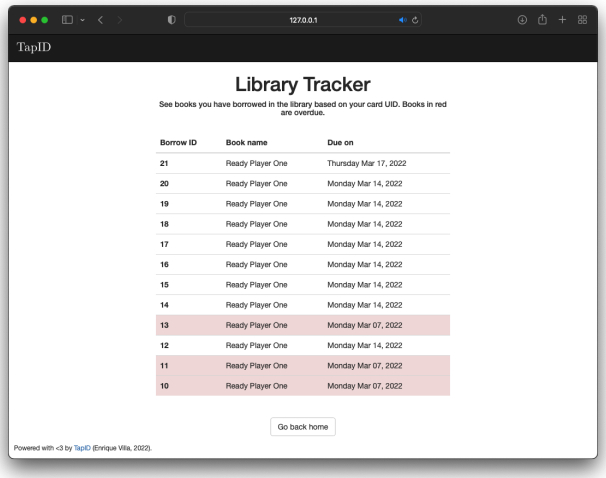


Figure 24: The table shown when the card has books associated with it's UID.

3) Attendance plug-in: The attendance plug-in fits in 17 lines of code, which showcases that plug-ins are not required to be massive in size and can be simple in nature. It simply updates a database with the time the person has come in (which in the plug-ins case is the time now), and returns the time now to give the Ground Module a date and time to display to the person logging in. The source code for this plug-in can be viewed in Appendix G and in the GitHub repository (see Appendix B).

E. Ground Module

1) Parts: The Ground Module uses the Raspberry Pi Pico, a microcontroller that houses an RP2040 processor that runs at 16MHz. The are three main reasons behind the Pi Pico's usage in the Ground Module. It has two UART, SPI, and I2C communication lines, meaning that the microcontroller can support a lot of peripherals, allowing for even further expandability. Aside from that, it supports MicroPython, a subset of Python that allows it to be usable in microcontrollers, and the micrcontroller has a total of 28 general purpose input/output pins (GPIO). The Pi Pico does not have Wi-Fi support built-in, so an ESP-01 module was used. It is a board that houses an ESP-8266, and communication with the board is done through AT commands (a format to send parseable commands) sent through UART. There were no libraries written for the board as MicroPython is not a very large community, so to ease TapID future development an ESP-01 utility class was created for TapID. It supports only basic commands, such as `ESP8266.startup()`, and the POST and GET HTTP protocols, but the class is enough to support TapAPI. The source code for the utility class can be seen in the ground_module_python/utils directory. The utility class holds methods displayed on Table V.

Name	Return
`change_uart_timeout(timeout)`	None
`send(cmd[, timeout])`	str
`startup([timeout])`	str
`wifi_mode(mode[, timeout])`	str
`connect_to_wifi(ssid, password[, timeout])`	str
`get_local_ip_address(timeout)`	str
`set_multiple_connections(z[, timeout])`	str
`establish_connection(type, ip, port[, timeout])`	str
`get(ip, route)`	str
`post(ip, route, payload)`	str

Table V: UART ESP-01 utility class.

The simplest Ground Module only requires the Raspberry Pi Pico and the ESP-01, but in order to display text — and in turn feedback — to users, a 16x2 I2C LCD was used and in order to accept user input, a 4x4 keypad was connected to the microcontroller.

2) Wiring and PCB: The Ground Module wiring was created in EasyEDA, a free web-app PCB schematic designer. Due to the sheer amount of connections, a small breadboard will not be ideal. In the initial prototyping stage, the different modules of the Ground Module were connected together using Dupont (jumper) cables and a breadboard. Once the wiring and code has been finalized and confirmed as working, a schematic was made. The schematic can be seen in Figure 25.

Once the wiring and modules have been finalized, a PCB design seen in Figure 26 was created using EasyEDA's auto-route feature. Note that auto-route is not always correct, and a preliminary check of the traces were done before sending the design to JLCPCB (a China-based PCB manufacturer

that manufactures PCBs for an affordable price) for manufacturing. The FR-4 base material 2-layer PCB uses a black HASL (with lead) finish, and is sized 63.8 mm x 49.8 mm. The silkscreen color was chosen to be white.

The Ground Module's PCB arrived 6 days after delivery via FedEx International Packet shipping. After a cursory view and some initial part fit testing, it appeared that the holes for the male JST connectors for the LCD, Keypad, and MFRC522 were of the incorrect pitch. Due to that, the initial prototype for the Ground Module hosts an inverted connector design, where instead of connecting the peripheral to the PCB, the PCB is connected to the peripheral. In future recreations, order the second iteration (see Figure 27) of the PCB with fixed connector pitches, better Raspberry Pi Pico positioning and better pin labeling. The files are also available in the repository.

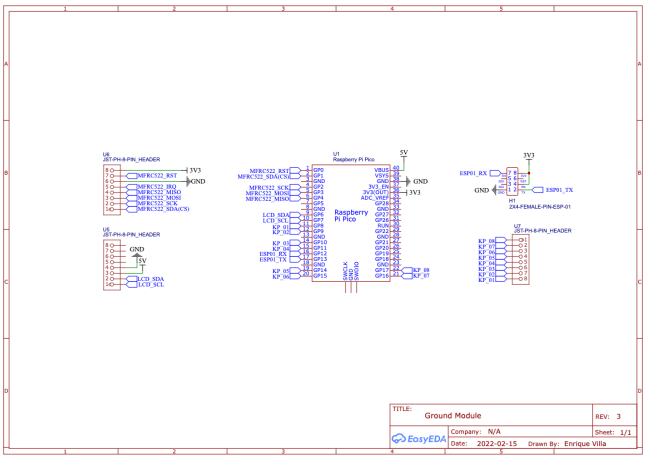


Figure 25: The Ground Module schematic.

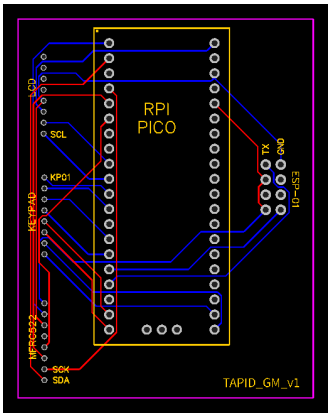


Figure 26: The TapID Ground Module PCB design.

3) Casing: The case was designed in Autodesk Fusion 360 and the .3mf file is available in the GitHub repository. It was designed with simplicity in mind, but it must look unique enough to be able to be easily recognized by students. The blueprint uses millimeters as its unit and can be seen in Figure 28. The top slides in and interlocks with the bottom body via the notches present on the cover's bottom. The

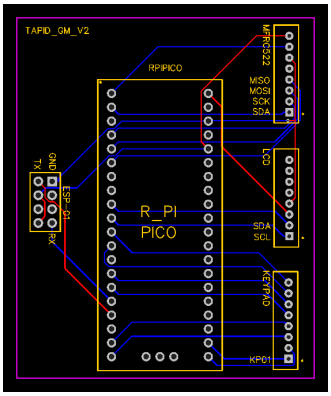


Figure 27: The second revision of the Ground Module PCB design with fixed pitches and improved silkscreen information.

lock between the top cap and bottom body is strong, as the case was 3D printed and the layers latch in between one another, creating a strong grip. A non 3D-printed case may not have the same grip, so an alternate design might need to be created.

The model was then sliced in Ultimaker Cura using a modified version of the program's default PLA settings. A 0.2mm layer height was used, 215C initial printing temperature (to help bed adhesion) and then 210C. The model used the "grid" infill with 3 layers of walls. The bed temperature was initially 65C then lowered to 60 after the first layer has been printed. Some supports have been used to print the overhangs for the USB cable and ventilation hole. The filament used was the gray eSUN PLA+ filament with a skirt before printing to encourage filament flow. PLA was used as it is biodegradable under certain conditions [22] and is the cheapest and most accessible and is safe to print compared to other 3D printer filaments [23]. The cap and the body were printed separately. See Figure 29 for the sliced model of the bottom body.

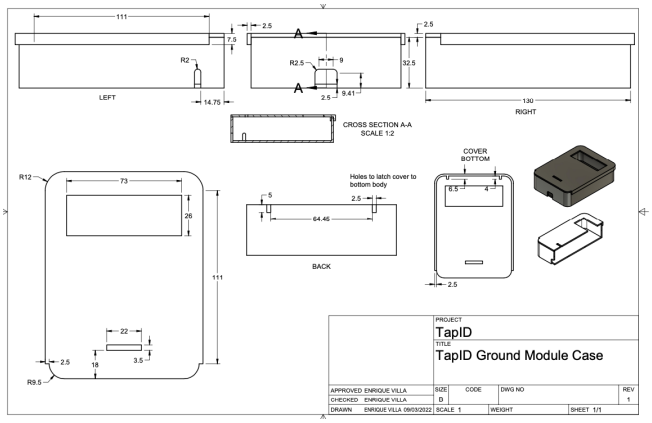


Figure 28: The Ground Module case blueprint.

4) Code: The MicroPython code was created in the PyCharm IDE using the MicroPython plug-in. The Ground Module always runs the main.py file flashed on its root

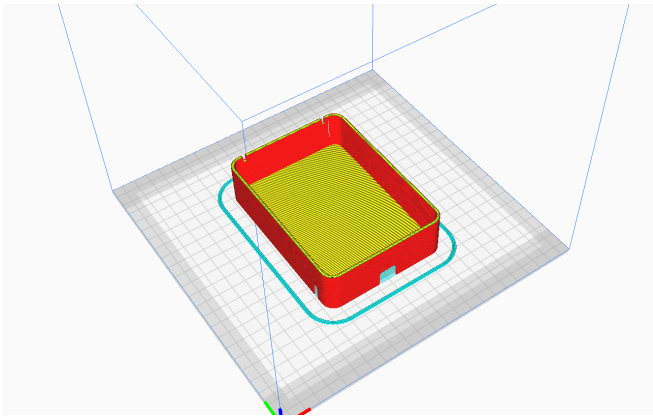


Figure 29: The sliced case model (bottom body).

directory. MicroPython automatically adds the Machine library, which holds all the hardware related libraries, such as UART, I2C, and Pin for controlling the GPIO. The code required for a working Ground Module from system boot to when it is ready is approximately 270 lines. The ground module comes with 3 example mains in the ground_module_python directory: template_main.py, library_main.py, and canteen_main.py. All three essentially share the same system boot code and only differs in what the payload sent to TapAPI is and how it handles different HTTP response codes. The template_main.py file is special as it holds an extra section for writing JWTs to the card. All three files are in the ground_module_python directory in the repository and template_main.py can also be seen in Appendix H.

A JWT has 16 sectors, each with 4 blocks that hold 16 bytes each, which total to 1024 bytes (or 1KB), but only around 724 bytes are usable since other blocks hold data such as access bytes, read/write keys, and the card's UID. The JWT used for testing used 544 bytes and took 34 sectors to write which indicates that 1KB is more than enough for a user's JWT, and there is still some space. The Ground Module knows to stop reading when it encounters the ASCII ETX character, `0x03` or 3 in decimal. If an ETX was not read, it stops reading immediately.

The Ground Module boots and then does a sample GET request at the servers root (/) route. The ESP-01 returns the HTTP packet as plaintext, meaning a parser had to be written to parse through HTTP responses for code to dynamically access the data received from the server. The parser (`parse_http_payload()` located in utils.py) uses standard Python string modification functions and the ujson library. The ESP8266 utility class custom-written for TapID sends an AT command through UART bus 0 at 115200 baud, then waits for a response by sending "AT", an AT command used for testing. If the ESP-01 is busy, it will not return anything. The function reads 1 character and checks if it is empty, and repeats until a complete message is received.

5) API response times: The API takes less than a second to respond, although the bottleneck lies in the Ground Module due to the unforeseen consequences that come with using UART to communicate and read packets. The module takes an average of 4.2 seconds (4280ms) to receive a response from the API (see Table VI). This time could be equivalent to that of paying with real chits so an improvement in this Raspberry Pi Pico to ESP-01 communication is required. But even if the ESP-01 is not fixed, some might find it a small price to pay to fix pollution, duplicate creation of chits, have a log of all transactions, and to completely remove human error in turn increasing reliability and sustainability for the canteen economy of schools. The boot times, and what the time needed in boot is for, is shown in Table VII.

Try #	Time (ms)
1	4288
2	4279
3	4273
Avg:	4280 (4.2s)

Table VI: Response times for Ground Module tested on the canteen Ground Module.

Part in boot sequence	**Time (ms)**	**AT command**
UART, I2C bus instantiation, initial ESP-01 handshake and setup	1447	`AT`
Configure ESP-01 to station mode (i.e., connect to Wi-Fi instead of making own network)	1010	`AT+CWMODE`
Connecting to a Wi-Fi network	11017	`AT+CWJAP`
Storing device IP and MAC addr	1016	`AT+CIFSR`
Disable multiple connection mode on ESP-01	1442	`AT+CIPMUX`
Establish TCP connection with API server	1021	`AT+CIPSTART`
Send an HTTP GET request on root API to obtain API version	2977	`AT+CIPSEND`
Total:	19930 (19.9s)	

Table VII: The Ground Module boot sequence and times.

6) Pricing: Table VIII lists down all the parts, payments, and prices for one Ground Module. The price can be improved by doing multiple things. Buying parts in bulk to pay one shipping fee for multiple items will improve prices. A local PCB manufacturer must be found as JLCPCB — the China-based PCB manufacturing service — has an extra FedEx processing fee of 576.80php even after the initial 700php shipping fee for 5. Buying PCBs in larger amounts

might allow the price to break even and in turn not make the manufacturing affect the price.

An estimate for the 3D printing will only assume that the print takes one try and no time is wasted in between printing the bottom body and the cover. The printer used for the project, the Creality Ender 3 v2, has a rating of 0.125kWh [24]. The print takes an estimate of 7 hours and 34 minutes and around 56g of filament, which means electricity-wise, the print cost approximately 10 pesos at the latest Meralco kWh rate (10.83 per kWh). The price of a 1kg spool of eSUN PLA+ filament is 750php, meaning the 56g used cost around 75 pesos. In total, the case would approximately cost 100-150 pesos.

Item	Price (PHP)	Description
Raspberry Pi Pico	254.75*	The microcontroller used for the project.
ESP8266 ESP-01	148*	Adds Wi-Fi functionality to the microcontroller.
4x4 Keypad	98*	Adds a user interface device to the Ground Module.
16x2 I2C LCD	149*	Allows the Ground Module to show messages.
8-pin JST connectors (x3)	30	Connects peripherals (i.e., the Keypad and the LCD) to the main PCB.
PCB manufacturing fee	25	–
PCB shipping charge	150	–
Total:	854 PHP	

Table VIII: The total pricing of a single Ground Module. Items with an asterisk have a shipping fee factored in and assume that you only buy one of the items.

V. Physical Results

The physical results are shown in the multiple figures in this section.

VI. Conclusion

Banknotes are bills used to pay in country with a currency system-based economy [1]. The banknotes due to it's physical nature unfortunately create a new efficient medium for pathogen to spread through [2]. Schools allow children to use banknotes as a currency within the system which creates a sanitary problem. Some schools try and counter the problem by creating chits, yet the plastic on the chits create an even better surface for pathogens to stick to [25]. TapID is a system that creates a "digital fingerprint" that create a secure way for individuals to prove their identity, which then can be used to authenticate transactions. A digital fingerprint can be used outside of authenticating transactions, as it can be used to facilitate authenticated interactions (i.e., interactions that require one or more parties to authenticate

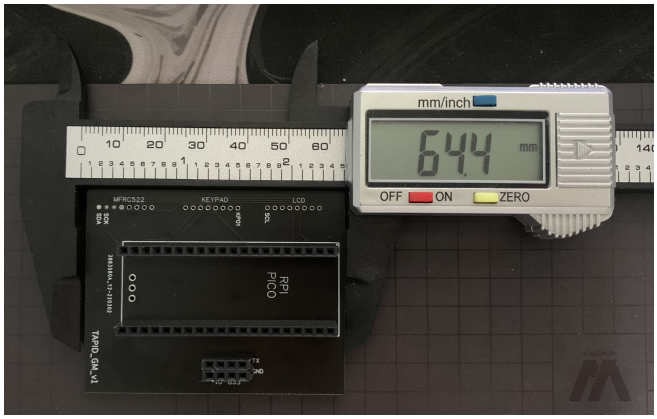


Figure 30: A measurement of the Ground Module PCB.

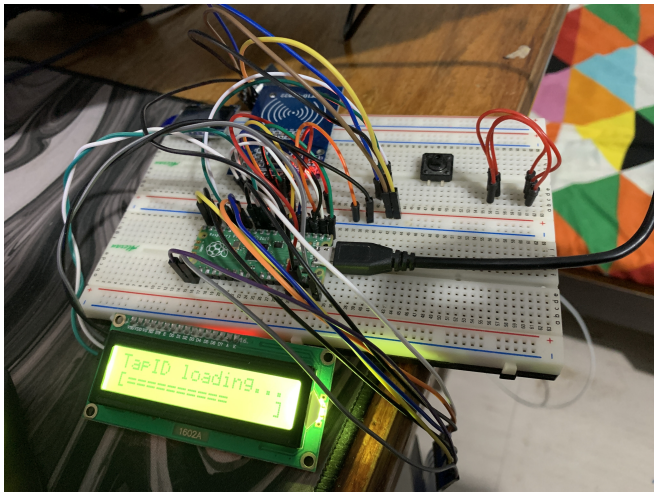


Figure 31: The breadboard prototype of a TapID Ground Module. This emphasizes the amount of connections, and the need for a PCB/protoboard.

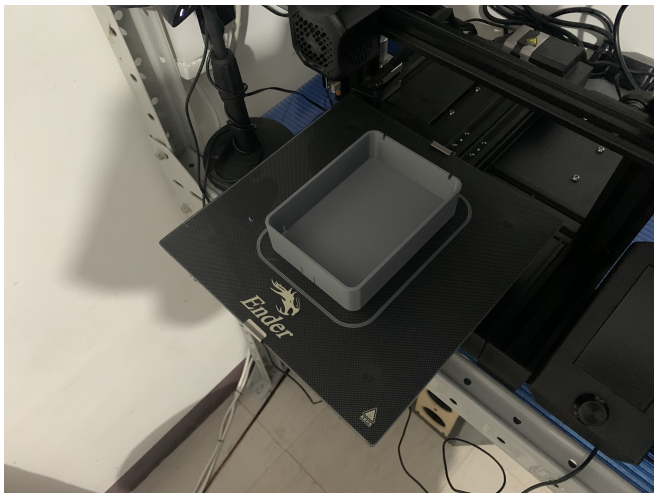


Figure 32: The bottom body of the Ground Module immediately after it finished printing.

Figure 33: Side view of the Ground Module.

Figure 34: The interior wiring of the Ground Module.

their identity) such as borrowing a book and signing.

Preliminary surveys show that there is a need for the the traditional systems to be replaced, as they are either too slow on unsanitary. TapID was built from the beginning to be expandable, allowing future researchers to improve on the system. It utilizes a Python Flask API for authentication, allowing users to do more than just respond using HTTP requests, but serve a website or a file. It was built using Python as Python has a thriving community with integrations for multiple services (such as Google), increasing the number of possible plug-ins. To even further promote future development, the code is available open-sourced on GitHub, a code storage/hosting platform, with a wiki to guide beginners in understanding the project. The system hypothetically should be better (in terms of cleanliness, efficiency, and data gathering) than a traditional system ran without electronics, but a school might have to pay more than a traditional system to get TapID implemented in its campus.

VII. Recommendations

The preliminary survey must be performed on faculty and staff to ensure that every canteen customer is covered in the data analysis. The measure of ”unsanitary” must also be scientifically done, as a simple Likert scale will not accurately give a numerical representation of cleanliness. Future studies could also determine if the payment in deploying the TapID ecosystem is more expensive than a normal deployment of the traditional methods, and if there is an extra overhead in terms of payments, determine if schools would be willing to pay the extra costs for the extra benefits provided. When deploying the system in a real-life situation, remember the details, improvements, and recommendations in the following sub-sections of this section.

Figure 35: Top view of the Ground Module.

A. Deployment

1) Network: Make sure that the server is connected on the same network as the Ground Modules, otherwise it will not be able to perform any without port forwarding. A Local Area Network is required, but a Wide Area Network is not, as all the system needs to authenticate is a local connection to the server. The server can be located offshore, but proper port forwarding and load balancing is needed (use a system like Kubernetes).

When deployed in an environment where hackers are an absolute must to defend, make sure to create an HTTPS certificate for the server to prevent packet sniffing. The preferable set-up for any small institution is a secondary network just for Tap-related traffic with an HTTPS (not HTTP) server to increase the server’s security. The secondary network does not need a fast internet connection (a 1mbps one would handle all traffic).

2) Maintenance: A person who has adequate knowledge in networking, Python programming, and electronics, must be assigned as the point-person for any errors in the server. At the system's current state, multiple errors are definitely still to be discovered and to be caught properly to ensure that a single error will not crash the entire system. The system also possibly has multiple vulnerabilities at this beta stage. If no person is adequate enough to watch over the system, personnel costs must also be factored in as a person must be hired to watch over.

3) TapAPI: Deploy the server on an Ubuntu Linux system (because Linux uses the least amount of resources and is a great learning experience) if there is a dedicated computer for handling the server. If none, a Raspberry Pi should be sufficient to handle authentication and plug-ins, even to external servers. The Linux server should have SSH enabled to allow easy access to the terminal. A `tmux` screen can be used to run the server in the background. Run `crontab -e` and link to a `.sh` (make sure to run `chmod +x` on the file to mark it as executable) file that runs the `tmux` screen immediately on boot for automatic on during a short/long power outage. A UPS would also increase the server's uptime. A guide is available on the TapID wiki about deployment, along with an entire list of beginner-friendly guides on "getting started", plug-in creation and TapAPI modification.

Once the server load starts to reach levels that the computer cannot handle, consider looking into load balancers like Nginx, Apache, or Kubernetes. To deploy the server, install the packages using `pipenv`. If you unsure, follow the wiki.

For aesthetics, you can use the 01-tapapi shell file and put it in the `/etc/update-motd.d/` directory for a cool log-in screen like in Figure 36. Make sure to mark the file as executable (`chmod +x`).

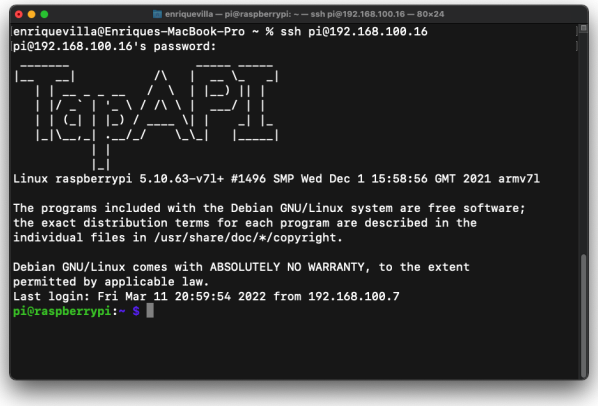


Figure 36: SSH TapAPI log-in screen using the 01-tapapi.sh file.

B. JWTs and encryption

The code in Appendix D can be modified to instead read a file filled with delimited JWTs and write them one by one. This way the writing is automated, in turn increasing efficiency when deploying the server for the first time in a school year. The settings used to create the RSA encryption keys are created using the industry standard, but a mathematically-inclined researcher could possibly put some further research into the security or efficiency of the implementation, as some of its parameters (such as the public exponent) could still be optimized.

C. Ground Module

There is an unusual issue where the `ESP8266` utility class takes longer than expected to receive a response from the authentication API. It is unsure where the issue lies, but it could possibly be with the read while(1) loop and the UART timeout. This issue can be looked into and fixed. A possible solution is to communicate with an NodeMCU ESP8266 via Serial and let it handle all the networking.

The Ground Module also is in a fairly unreliable state, as multiple errors still have to be debugged and caught properly in a `try ... except` bracket. One example of such error is at the HTTP parser function as the server often returns malformed or unsupported (i.e., not implemented in code not in the Hyper Text protocol itself) HTTP packets that the parser (`parse_http_payload()`) does not know how to handle.

1) PCB: The Ground Module's PCB also has to be replaced with the second revision (available in the repository; see Figure 27). Due to the time constraint, there was not enough time to create a secondary version of the PCB where the JST male connector header pin pitches are correctly spaced. If the TapID Ground Module were to use more amperage than the PCB traces can handle to send power to a peripheral, consider looking into optocoupler-enhanced relays instead to prevent overcurrent damage to the microcontroller or any other component.

2) Attachments: The Ground Module could use better ways to attach the MFRC522 and LCD to the cover of the case. Screws or correctly placed ledges could work in securely attaching the LCD and MFRC522. In the prototype shown in Figure 34, duct tape was used to hold the modules, which in practice could be better.

3) Power backups: A simple power bank could power a Raspberry Pi 3 for an estimate of more than an hour depending on the power banks mAh rating. If a computer is used as the server, look into universal power supplies (UPS) instead. Future studies must determine the ideal sized battery for power outages. One could possibly study the frequency of power outages in a certain campus and create an estimate based on that.

APPENDIX A
TAPID PRELIMINARY SURVEY

The TapID preliminary survey can be fully viewed at the following URL: https://drive.google.com/file/d/

16O9Mfx-tPaKaF0VRF2U9m0iVARQxJeP_/view?usp=sharing

APPENDIX B
TAPID GITHUB REPOSITORY

The TapID GitHub repository is available at the following URL: https://github.com/iggyvilla/TapID

APPENDIX C
TAPID WIKI

The TapID beginner-friendly wiki can be viewed at the following URL: https://github.com/iggyvilla/TapID/wiki/Before-you-start...

APPENDIX D
TAPID JWT GENERATION PYTHON SCRIPT

```
import jwt
from cryptography.hazmat.primitives.asymmetric
    import rsa
from cryptography.hazmat.primitives import
    serialization

# 65537 as a public exponent is standard (e=3 is
    used sometimes, but a
#  student can do future research on the security
    of this RSA implementation)
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048
)

# What's all this PEM, PKCS8, PKCS1 stuff?
# Read: https://stackoverflow.com/questions
    /48958304/pkcs1-and-pkcs8-format-for-rsa-
    private-key
encrypted_pem_private_key = private_key.
    private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.
        NoEncryption()
)

encrypted_pem_public_key = private_key.public_key
    ().public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.PKCS1
)

# The private keys come in a byte-encoded format,
    so we turn it into a string by using .decode('
    utf-8')
# Use these encoded keys to make it possible to
    save it in a database
# We can throw away the private key unless needed
    in testing
# utf8_pem_private_key = encrypted_pem_private_key
    .decode('utf-8')
utf8_pem_public_key = encrypted_pem_public_key.
    decode('utf-8')

# The encoded variable can then be stored in the
    RFID card
encoded = jwt.encode(
    {
        "name": "Enrique␣Miguel␣Jose␣E.␣Villa",
        "grade": 12,
        "pass": "Hedcen92"
    },
    encrypted_pem_private_key,
    algorithm="RS256"
)
```

APPENDIX E
TAPAPI CANTEEN PLUG-IN

```
from utils.responses import PluginResponse
from utils.parse_payload import
    TapAPIRequestPayload
from plugins.plugin_utils import canteen_utils

event_name = "canteen_event"


def run(jwt_decoded: dict, payload_data:
    TapAPIRequestPayload, args, conn) ->
    PluginResponse:
    uid = payload_data.uid
    event_data = payload_data.event_data

    old_bal = canteen_utils.get_balance_of_uid(
        conn, uid)

    # If the action (subtract or add) is not in
        event_data
    if "action" not in event_data:
        return PluginResponse(400, payload={"msg":
            "Invalid␣dict"})

    if event_data["action"] == "subtract":
        new_bal = old_bal - event_data["bal"]
    elif event_data["action"] == "add":
        new_bal = old_bal + event_data["bal"]
    else:
        return PluginResponse(400, payload={"msg":
            "Invalid␣action"})

    # If there isn't enough balance
    if new_bal < 0:
        return PluginResponse(response_code=403,
            payload={"msg": "missing␣balance"})
    else:
        canteen_utils.update_balance_of_uid(conn,
            uid, new_bal=new_bal, action=
            event_data["action"], bal=event_data["
            bal"])
        return PluginResponse(response_code=200,
            payload={"msg": "success", "new_bal":
            new_bal})
```

APPENDIX F
TAPAPI LIBRARY PLUG-IN

```
from plugins.plugin_utils import library_utils
from utils.parse_payload import
    TapAPIRequestPayload
import os
import logging
import psycopg2.errors
from utils.responses import PluginResponse
from googleapiclient.discovery import build
from googleapiclient.errors import HttpError
from google.oauth2.credentials import Credentials
from google.auth.transport.requests import Request
from google_auth_oauthlib.flow import
    InstalledAppFlow


event_name = "borrow_book"
_log = logging.getLogger("main.logger")
```

```
# If modifying these scopes, delete the file token
    .json.
SCOPES = ['https://www.googleapis.com/auth/
    spreadsheets']

# Sheets API code from https://developers.google.
    com/sheets/api/quickstart/python
# Edit with spreadsheet ID
SPREADSHEET_ID = '12
    mlNwnZBP7GxGX5OJ0oCd19DcYDErBRVncyljzSwiwI'
# Edit with the length of the first row of headers
RANGE_NAME = 'borrow_log!A1:D1'

creds = None

# The file token.json stores the user's access and
     refresh tokens, and is
# created automatically when the authorization
    flow completes for the first
# time.
if os.path.exists('plugins/token.json'):
    creds = Credentials.from_authorized_user_file(
        'plugins/token.json', SCOPES)

# If there are no (valid) credentials available,
    let the user log in.
if not creds or not creds.valid:
    if creds and creds.expired and creds.
        refresh_token:
        creds.refresh(Request())
    else:
        flow = InstalledAppFlow.
            from_client_secrets_file(
            'plugins/credentials.json', SCOPES)
        creds = flow.run_local_server(port=0)
    # Save the credentials for the next run
    with open('plugins/token.json', 'w') as token:
        token.write(creds.to_json())


def run(jwt_decoded: dict, payload_data:
    TapAPIRequestPayload, args, conn) ->
    PluginResponse:

    # Extract the payload data
    data = payload_data.event_data

    # Insert an entry into the database
    try:
        db_entry = library_utils.borrow_book(
            uid=payload_data.uid,
            book_id=data['book_id'],
            due_on=data['due_on'],
            conn=conn
        )
    except (KeyError, psycopg2.errors.
        InvalidForeignKey):
        return PluginResponse(response_code=400,
            payload={"msg": "invalid␣data"})
    except psycopg2.errors.Error:
        return PluginResponse(response_code=500,
            payload={"msg": "internal␣server␣error
            "})

    # If the client requests you save to Google
        Docs
    if data['save_to_google_docs']:
        try:
            # Build the Google service with your
                credentials
            service = build('sheets', 'v4',
                credentials=creds)

            # Call the Sheets API
            sheet = service.spreadsheets()
            #               Borrow ID    Book
                Title   Student Name         Due
                When
            sheets_entry = (db_entry[0], db_entry
                [1], jwt_decoded["name"], db_entry
                [2].strftime("%A␣%b␣%d,␣%Y"))
            result = sheet.values().append(
                spreadsheetId=SPREADSHEET_ID,
                range=RANGE_NAME,
                valueInputOption="USER_ENTERED",
                insertDataOption="INSERT_ROWS",
                body={
                    "values": [sheets_entry]
                }
            ).execute()

            if result:
                return PluginResponse(
                    response_code=200, payload={"
                    google_resp": result, "entry":
                     sheets_entry})
            else:
                return PluginResponse(
                    response_code=500, payload={"
                    msg": "google␣api␣error"})
        except HttpError:
            return PluginResponse(response_code
                =500, payload={"msg": "saved␣to␣
                local␣db␣but␣cant␣connect␣to␣
                google␣api"})

    return PluginResponse(response_code=200,
        payload={"msg": "ok"})
```

APPENDIX G
TAPAPI ATTENDANCE PLUG-IN

```
from utils.responses import PluginResponse
from utils.parse_payload import
    TapAPIRequestPayload
from plugins.plugin_utils import attendance_utils
from datetime import datetime
import psycopg2.errors


event_name = "attendance"


def run(jwt_decoded: dict, payload_data:
    TapAPIRequestPayload, args, conn) ->
    PluginResponse:
    try:
        attendance_utils.
            insert_into_attendance_logs(
            jwt_decoded["name"], payload_data.uid,
             conn)
        # Format time to Day (name) Day (number)
            Hour (24-hour):minute
        return PluginResponse(200, payload={"msg":
             "ok", "data": {"time_now": datetime.
            now().strftime("%a␣%d␣%H:%I")}})
    except psycopg2.errors.Error:
        return PluginResponse(500, payload={"msg":
             "internal␣server␣database␣error"})
```

APPENDIX H
GROUND MODULE SOURCE CODE

```
"""
TEMPLATE CODE
"""
```

```
from machine import I2C, Pin, UART
from utils import parse_at_response,
    jwt_to_rfid_array, formatted_uid, ESP8266,
    writable_rfid_blocks, parse_ip_response
import utime
from sys import exit
from mfrc522 import MFRC522
from pico_i2c_lcd import I2cLcd

# To keep track of previous RFID card
previous_card = []

# Toggle between read/write modes
# If write is True, then it will write the JWT in
    "jwts.txt"
# The code can be modifiable to instead write JWTS
     in an array one by one
WRITE = False

# MFRC522 reader class instantiation
reader = MFRC522(spi_id=0, sck=2, miso=4, mosi=3,
    cs=1, rst=0)

# Instantiate I2C communications for LCD
i2c = I2C(1, sda=Pin(6), scl=Pin(7), freq=400000)
I2C_ADDR = i2c.scan()[0]
lcd = I2cLcd(i2c, I2C_ADDR, 2, 16)

# Instantiate UART communications with ESP8266
uart1 = UART(0, tx=Pin(12), rx=Pin(13), baudrate
    =115200, timeout=5000)
esp8266 = ESP8266(uart1, tx_pin=Pin(12), rx_pin=
    Pin(13))
print(uart1)

# TapAPI IP and port
SERVER_IP = "192.168.100.7"
SERVER_PORT = 8000


def clear_loading_screen(percent_done):
    """Function to easily set the loading screen
        without repeating code"""
    lcd.clear()
    # Think of move_to like moving where the type
        cursor is. That's exactly what it's doing!
    lcd.move_to(0, 0)
    lcd.putstr("TapID␣loading...")
    lcd.move_to(0, 1)
    percent_done = percent_done if percent_done <=
         14 else 14
    lcd.putstr("[" + "=" * percent_done + "␣" *
        (14-percent_done) + "]")


def card_on_sensor_msg():
    """Function to easily put "enter card on
        sensor" message"""
    lcd.clear()
    lcd.putstr("Put␣card␣on")
    lcd.move_to(0, 1)
    lcd.putstr("sensor.")


# Test AT startup and see if ESP-01 is wired
    correctly
if esp8266.startup() is None:
    print("ESP8266␣not␣setup␣properly")
    exit()

clear_loading_screen(2)

# Set the Wi-Fi mode to station mode (connects to
    WiFi instead of creating its own WiFi signal)
esp8266.wifi_mode(ESP8266.MODE_STATION)

clear_loading_screen(3)

# Connect to an AP
esp8266.connect_to_wifi("Akatsuki␣Hideout", "8
    prongedseal", timeout=4000)

clear_loading_screen(4)

# Get the modules IP address for metric use
resp = esp8266.get_local_ip_address()
module_ip = parse_ip_response(resp)

clear_loading_screen(8)

# Make sure we only connect to one network (the
    ESP-01 can connect to multiple networks)
esp8266.set_multiple_connections(False)

clear_loading_screen(10)

# Establish a TCP connection to the HTTP server
conn_resp = esp8266.establish_connection("TCP",
    SERVER_IP, SERVER_PORT)

# See if server is offline
if parse_at_response(conn_resp) == "ERROR\nCLOSED\
    n":
    lcd.clear()
    lcd.move_to(0, 0)
    lcd.putstr("Can't␣connect␣to")
    lcd.move_to(0, 1)
    lcd.putstr("TapID␣server.")
    exit()

# Make a GET request to TapAPI to see if the
    server is responsive
payload = esp8266.get(SERVER_IP, "/")

clear_loading_screen(12)

lcd.clear()
lcd.move_to(0, 0)
lcd.putstr("API␣version:")
lcd.move_to(0, 1)
lcd.putstr(str(payload.body['api_version']))

utime.sleep(1)

# Message to show if the module is in WRITE mode
if WRITE:
    print("Ground␣Module␣in␣write␣mode.␣Writing␣
        JWT␣from␣jwts.txt")

print("SYSTEM␣INITIALIZATION␣COMPLETE\n\n")

card_on_sensor_msg()

while True:
    # Initialize the MFRC522 reader
    # Taken from example code from danjperron/
        micropython-mfrc522/Pico_examples/
        Pico_read.py
    reader.init()
    (stat, tag_type) = reader.request(reader.
        REQIDL)

    if stat == reader.OK:
        (stat, uid) = reader.SelectTagSN()

        if uid == previous_card:
            continue

        if stat == reader.OK:
            # Display that a card was detected
```

```
lcd.clear()
lcd.move_to(0, 0)
lcd.putstr("Card␣detected")

print(f"Card␣detected␣(uid={reader.
    tohexstring(uid)})")

previous_card = uid

# The default card key 0xFF six times
# For testing purposes you can use it
    to reduce possible headaches
# In deployment, write the key in the
    trailer sectors (where sector_n
    %4==3; 1,3,5,7...)
defaultKey = [255, 255, 255, 255, 255,
     255]

# Function to read entire card, remove
     comment if needed
# reader.MFRC522_DumpClassic1K(uid,
    Start=0, End=64, keyA=defaultKey)

# If you want to use the ground module
     to write JWTs to cards
if WRITE:
    jwt_arr = jwt_to_rfid_array("jwts.
        txt")
    print(jwt_arr)
    for x, block_num in enumerate(
        writable_rfid_blocks()):
        print(f"{x=}")
        status = reader.auth(reader.
            AUTHENT1A, block_num,
            defaultKey, uid)
        if status == reader.OK:
            try:
                if x == 33:
                    print(jwt_arr)
                    print(jwt_arr[x])

                status = reader.write(
                    block_num, jwt_arr
                    [x])
                print(f"{jwt_arr[x]=}"
                    )
                if status != reader.OK
                    :
                    print("unable␣to␣
                        write")
                    break
            except IndexError:
                break
        else:
            print("Authentication␣
                error␣for␣writing")
            break

        print(f"Block␣{block_num}␣
            success␣(#{x}).")

    print("Done␣writing.")

    exit()

# Buffer for storing card information
jwt_buf = []

# Boolean to keep track if entire JWT
    was read
read_complete = False

# writable_rfid_blocks() are all
    blocks minus the trailer sectors)
for block_num in writable_rfid_blocks
    ():
    status = reader.auth(reader.
        AUTHENT1A, block_num,
        defaultKey, uid)
    end_of_text = False

    if status == reader.OK:
        _status, read_block = reader.
            read(block_num)

        if status == reader.OK:
            for char in read_block:
                # If we reach the end
                    of text character
                    (3 in decimal)
                if char == 3:
                    end_of_text = True
                    read_complete =
                        True
                    break
                jwt_buf.append(chr(
                    char))
        else:
            print("Error␣reading␣RFID␣
                card")
    else:
        break

    if end_of_text:
        break

# If the end of text character wasn't
    read
if not read_complete:
    print("Card␣not␣read␣completely,␣
        ignoring.")
    card_on_sensor_msg()
    continue

# JWTs have two periods. This is the
    only not-so-resource-intensive way
     to
# Check if there is a valid JWT
if jwt_buf.count('.') != 2:
    print('Not␣a␣JWT.␣Ignoring␣card.')
    continue

print(f"Read␣a␣JWT␣from␣RFID.")

lcd.move_to(0, 1)
lcd.putstr("Card␣read!")

# A valid TapAPI payload
payload = {
    "jwt": "".join(jwt_buf),
    "uid": formatted_uid(uid),
    "event_name": "example_event",
    "event_data": {
        "hello": "world",
        "test": "data"
    }
}

print(payload)

# Establish a TCP HTTP connection to
    the server ip and port
esp8266.establish_connection(type="TCP
    ", ip=SERVER_IP, port=SERVER_PORT)

lcd.move_to(0, 1)
lcd.putstr("Authenticating")

# Perform the post request for
    authentication
```

```
            resp = esp8266.post(ip=SERVER_IP,
                route="/event", payload=payload)
            print("RECV:")
            print(resp)

            # If everything went well (200 OK)
            if resp.response_code == 200:
                # Display that everything went
                    well to the user, you can do
                    anything in this if statement
                lcd.clear()
                lcd.move_to(0, 0)
                lcd.putstr(f"Hello, G{resp.body['
                    jwt_decoded']['grade']}")
                lcd.move_to(0, 1)
                lcd.putstr(resp.body['jwt_decoded'
                    ]['name'].split(' ')[0])

            print("Done processing.\n")
        else:
            pass
    else:
        previous_card = []

    utime.sleep(1)
```

APPENDIX I
RAW PRELIMINARY SURVEY DATA

The raw preliminary survey can be viewed here: https://docs.google.com/spreadsheets/d/1fQyalsZKHNixsadLpp17c7499OdoBy6Zi4jxwHnQIo0/edit?usp=sharing

ACKNOWLEDGMENTS

The author of this paper would like to thank Daniel J. Perron (danjperron) and Tyler Peppy (T-622) for their open-sourced MicroPython libraries available on GitHub that allowed easy interfacing with the MFRC522 RFID module and 16x2 I2C LCD (respectively) for the Raspberry Pi Pico.

The following people that will be thanked are YouTube channels and have taught the author through their videos. Very special thanks to Corey Schafer and Tim Ruscica (otherwise known as TechWithTim) for teaching the author the much-needed information about Python and web development needed to build the entire project. For the information needed for the projects electronics, thank you to Andrei from Electronoobs, Scott from "GreatScott!", and Joe Barnard from BPS Space. For the 3d-printing and CAD design, thank you to Teaching Tech and Product Design Online.

REFERENCES

[1] M. Hall, "Barter system vs. currency system: What's the difference?" Sep 2021. [Online]. Available: https://www.investopedia.com/ask/answers/061615/what-difference-between-barter-and-currency-systems.asp

[2] H. Gedik, T. A. Voss, and A. Voss, "Money and transmission of bacteria," *Antimicrobial Resistance and Infection Control*, vol. 2, no. 1, 2013.

[3] G. Leibbrandt, *A billion here, a billion there: The statistics of payments*. Swift Institute, 2009. [Online]. Available: https://swiftinstitute.org/wp-content/uploads/2012/10/The-Statistics-of-Payments_v15.pdf

[4] A. K. Simon, G. A. Hollander, and A. McMichael, "Evolution of the immune system in humans from infancy to old age," *Proceedings of the Royal Society B: Biological Sciences*, vol. 282, no. 1821, p. 20143085, 2015.

[5] S. A. Dewanto, M. Munir, B. Wulandari, and K. Alfian, "Mfrc522 rfid technology implementation for conventional merchant with cashless payment system," *Journal of Physics: Conference Series*, vol. 1737, no. 1, p. 012012, 2021.

[6] K. Bonsor and W. Fenlon, "How rfid works," Nov 2007. [Online]. Available: https://electronics.howstuffworks.com/gadgets/high-tech-gadgets/rfid.htm

[7] S. Boboila and P. Desnoyers, "Write endurance in flash drives: Measurements and analysis," in *8th USENIX Conference on File and Storage Technologies (FAST 10)*. San Jose, CA: USENIX Association, Feb. 2010. [Online]. Available: https://www.usenix.org/conference/fast-10/write-endurance-flash-drives-measurements-and-analysis

[8] F. Attar, A. Bhatkar, S. H. Haider, and T. Kadiwala, "Rfid based universal transaction system," *Int. J. Sci. Res. in Computer Science and Engineering*, vol. 8, no. 2, p. 52–54, 2020. [Online]. Available: https://www.isroset.org/pub_paper/IJSRCSE/8-IJSRCSE-03359.pdf

[9] P. Christensson, "Database," 2009. [Online]. Available: https://techterms.com/definition/database

[10] E. F. Codd, "A relational model of data for large shared data banks," *Association for Computing Machinery*, vol. 13, no. 6, p. 377–387, Jun 1970. [Online]. Available: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf

[11] N. M. Iacob and M. V. Moise, "Centralized vs. distributed databases. case study," *Academic Journal of Economic Studies*, vol. 1, pp. 119–130, 2015.

[12] M. V. Caya, N. F. Losaria, M. J. Manzano, H. K. Tan, and R. V. Pellegrino, "Cashless transaction for resort club amenities using rfid technology," *2017IEEE 9th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM)*, p. 1–5, 2017.

[13] D. Murray, T. Koziniec, M. Dixon, and K. Lee, "Measuring the reliability of 802.11 wifi networks," in *2015 Internet Technologies and Applications (ITA)*, 2015, pp. 233–238.

[14] B. Siregar, N. Munawar, Seniman, and Fahmi, "Cashless payment system using rfid with nrf24l01 communication," in *2020 4rd International Conference on Electrical, Telecommunication and Computer Engineering (ELTICOM)*, 2020, pp. 186–190.

[15] "nrf24l01+ single chip 2.4ghz transceiver preliminary

product specification v1.0," Mar 2008. [Online]. Available: https://www.sparkfun.com/datasheets/Components/SMD/nRF24L01Pluss_Preliminary_Product_Specification_v1_0.pdf

[16] P. Christensson, "Api," 2016. [Online]. Available: https://techterms.com/definition/api

[17] "What is a rest api?" May 2020. [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api

[18] T. Solutions, "A complete guide to wan vs. lan vs. man: Differences," Nov 2019. [Online]. Available: https://techsolutions.cc/guide-wan-vs-lan-vs-man/

[19] "Introduction to json web tokens," Mar 2020. [Online]. Available: https://jwt.io/introduction

[20] G. Simmons, "Rsa encryption," 2009. [Online]. Available: https://www.britannica.com/topic/RSA-encryption

[21] D. Boneh, "Twenty years of attacks on the rsa cryptosystem," *American Mathematical Society (AMS)*, vol. 46, no. 2, p. 203–213, 1999. [Online]. Available: http://crypto.stanford.edu/~dabo/abstracts/RSAattack-survey.html

[22] J. O'Connell and A. Bohlooli, "Is pla actually biodegrable?" Sep 2021. [Online]. Available: https://all3dp.com/2/is-pla-biodegradable-what-you-really-need-to-know/

[23] D. Miller, "Should you worry about 3d printer fumes?" Mar 2021. [Online]. Available: https://io3dprint.com/should-you-worry-about-3d-printer-fumes/

[24] F. Arceo, "3d solved," 2021. [Online]. Available: https://3dsolved.com/ender-3-power-consumption/

[25] C. Newey, A. T. Olausson, A. Applegate, A. A. Reid, R. A. Robison, and J. H. Grose, "Presence and stability of sars-cov-2 on environmental currency and money cards," 2021.