In modern financial systems like Mobile Money (MoMo), APIs serve as the gateway to sensitive user data. Ensuring that only authorized personnel can access, modify, or delete transaction records is critical for maintaining trust and data integrity.

For this project, we implemented HTTP Basic Authentication. This security layer works by requiring the client to provide a set of credentials, in our case, a username (team5) and a password (ALU2025), before the server processes any request.

1. When a request is made, the server checks for an Authorization header.
2. If the header is missing or incorrect, the server responds with a 401 Unauthorized status and a WWW-Authenticate header, signaling the browser to prompt the user for credentials.
3. This prevents "open access" to the 1,691 transaction records, ensuring that the MoMo dataset is only visible to the intended developers or administrators.

This API provides a full suite of CRUD (Create, Read, Update, Delete) operations to manage the MoMo dataset. All endpoints require Basic Authentication.

- GET
- /transactions
- Fetches the entire collection of 1,691 records.
- A JSON array containing all transaction objects.

- GET
- /transactions/{id}
- Retrieves a specific record by its ID.
- This endpoint is powered by Dictionary Lookup (O(1)), providing the fastest possible retrieval speed.
- The JSON object for the requested ID.
- Returned if the ID does not exist.

- POST
- /transactions
- Adds a new record to the memory.
- Requires sender, amount, type, and body.
- Returns the new object with a system-generated ID.

- PUT
- /transactions/{id}
- Modifies existing fields (e.g., updating an amount or sender).
- Uses Linear Search (O(n)) to locate the record within the list structure before applying updates.
- The updated transaction object.

- DELETE
- /transactions/{id}
- Permanently deletes a record.
- The system removes the entry from both the list and the dictionary to maintain data synchronization.
- {"message": "Deleted"}

We implemented two ways to find a transaction by its internal_id:

1. Scans the entire list from beginning to end until it finds the matching ID (or reaches the end).

```python
def linear_search(transactions, target_id):
    for tx in transactions:
        if tx["internal_id"] == target_id:
            return tx  # or just break if you only need to find it
    return None
```

```python
# Linear search
start = time.perf_counter()
for sid in search_ids:
    for tx in transactions:
        if tx["internal_id"] == sid:
            break
linear_time = time.perf_counter() - start
```

2. For Dictionary Lookup we used a Python dictionary (hash table) where the key is the internal_id and the value is the full transaction dictionary. Lookup is done with .get().

```python
# Build dictionary for fast lookup
tx_dict = {tx["internal_id"]: tx for tx in transactions}
```

```python
# Dictionary lookup
start = time.perf_counter()
for sid in search_ids:
    _ = tx_dict.get(sid)
dict_time = time.perf_counter() - start
```

For

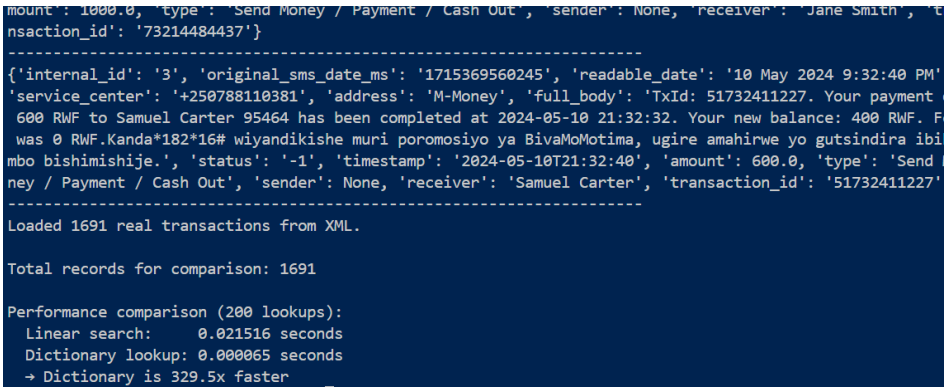
We measured the time taken to perform on the real dataset of .

The Code we used for comparison:

```
import time
import random
```

```
# Build dictionary for fast lookup
tx_dict = {tx["internal_id"]: tx for tx in transactions}

# Select 200 random existing IDs to search
if len(transactions) < 200:
    search_ids = [tx["internal_id"] for tx in transactions]
else:
    search_ids = random.sample([tx["internal_id"] for tx in transactions], 200)

random.shuffle(search_ids)

# Linear search
start = time.perf_counter()
for sid in search_ids:
    for tx in transactions:
        if tx["internal_id"] == sid:
            break
linear_time = time.perf_counter() - start

# Dictionary lookup
start = time.perf_counter()
for sid in search_ids:
    _ = tx_dict.get(sid)
dict_time = time.perf_counter() - start

print(f"\nPerformance comparison ({len(search_ids)} lookups):")
print(f"  Linear search:     {linear_time:8.6f} seconds")
print(f"  Dictionary lookup: {dict_time:8.6f} seconds")
```

We chose 200 lookups because the measured time is more larger & stable, and would give us a clear comparison.



Our dictionary is ~400 times faster for 200 lookups. Some of the reasons we attribute to that are:

has time complexity in the worst and average case. It must compare the target ID against every single record in the list.

has average time complexity. Python dictionaries use a hash table:

1. The key (internal_id) is passed through a hash function → produces a number
2. That number is used as an index to jump almost directly to the right memory location
3. Very few comparisons are needed (usually 1 or 2)

Other Data structures we could use for this are:

Binary Search	O(log n)	If you keep the list sorted by ID	Requires sorting (O(n log n) initially)

B-Tree / Balanced BST	O(log n)	Very large datasets, disk-based storage	More complex to implement
Trie (for string IDs)	O(k) where k = length of key	If IDs are long strings with common prefixes	Higher memory usage
Database Index (e.g. SQLite)	~O(log n) or better	Persistent data, millions of records	Needs external database

The recommendation we see best for this project here is a dictionary (hash table). It is simple code, has excellent average-case performance and is built into Python with a memory usage that is acceptable for 1691 records.

Incase someone wants to switch we recommend that they need to search by multiple fields (then consider database) and also they need range queries (e.g. all transactions between two dates → sorted list + binary search or database)

While allowed us to secure the MoMo API quickly, it has significant weaknesses in a real-world financial environment:

Basic Auth credentials are sent as a Base64 string. ; it is a reversible encoding format. If a hacker intercepts the network traffic (a "Man-in-the-Middle" attack), they can decode the string in milliseconds to steal the *team5:ALU2025* credentials.

With Basic Auth, the client (browser or Postman) must send the username and password with . This increases the risk of exposure. Unlike modern systems, there is no built-in way to "log out" or revoke a session without closing the entire browser.

Since the credentials do not change, an attacker who captures a single request header can "replay" it later to gain unauthorized access. There is no "nonce" or timestamp to verify the freshness of the request.

For a production-grade MoMo API, we would recommend:

- Allows for short-lived sessions that expire automatically.
- Separates authentication from authorization, common in banking APIs.
- Mandatory to encrypt the communication channel so that even Basic Auth headers cannot be read by third parties.