

Task 5 – Data Structures & Algorithms (DSA Integration)

Introduction to API Security

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.

How it Works in our System:

- 1. The Handshake:** When a request is made, the server checks for an Authorization header.
 - 2. Credential Verification:** 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. Data Protection:** This prevents "open access" to the 1,691 transaction records, ensuring that the MoMo dataset is only visible to the intended developers or administrators.
-

Documentation of Endpoints

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

1. Retrieve All Transactions

- **Method:** GET
- **URL:** /transactions
- **Description:** Fetches the entire collection of 1,691 records.
- **Response (200 OK):** A JSON array containing all transaction objects.

2. View Single Transaction

- **Method:** GET
- **URL:** /transactions/{id}
- **Description:** Retrieves a specific record by its ID.
- **DSA Note:** This endpoint is powered by Dictionary Lookup ($O(1)$), providing the fastest possible retrieval speed.
- **Response (200 OK):** The JSON object for the requested ID.
- **Error (404 Not Found):** Returned if the ID does not exist.

3. Create New Transaction

- **Method:** POST
- **URL:** /transactions
- **Description:** Adds a new record to the memory.
- **Request Body (JSON):** Requires sender, amount, type, and body.
- **Response (201 Created):** Returns the new object with a system-generated ID.

4. Update Record

- **Method:** PUT
- **URL:** /transactions/{id}
- **Description:** Modifies existing fields (e.g., updating an amount or sender).
- **DSA Note:** Uses Linear Search ($O(n)$) to locate the record within the list structure before applying updates.
- **Response (200 OK):** The updated transaction object.

5. Remove Record

- **Method:** DELETE
- **URL:** /transactions/{id}
- **Description:** Permanently deletes a record.
- **Logic:** The system removes the entry from both the list and the dictionary to maintain data synchronization.
- **Response (200 OK):** {"message": "Deleted"}

Implementation

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

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

```
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
```

```
# 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().

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

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

For Measurement & Comparison

We measured the time taken to perform **200 random lookups instead of 20** on the real dataset of **1691 transactions**.

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.

Results:

```
amount': 1000.0, 'type': 'Send Money / Payment / Cash Out', 'sender': None, 'receiver': 'Jane Smith', 'transaction_id': '73214484437'}
-----
{'internal_id': '3', 'original_sms_date_ms': '1715369560245', 'readable_date': '10 May 2024 9:32:40 PM', 'service_center': '+250788110381', 'address': 'M-Money', 'full_body': 'TxId: 51732411227. Your payment of 600 RWF to Samuel Carter 95464 has been completed at 2024-05-10 21:32:32. Your new balance: 400 RWF. Fw was 0 RWF.Kanda*182*16# wiyandikishe muri poromosiyo ya BivaMoMotima, ugire amahirwe yo gutsindira ibi mbo bishimishije.', 'status': '-1', 'timestamp': '2024-05-10T21:32:40', 'amount': 600.0, 'type': 'Send Money / Payment / Cash Out', 'sender': None, 'receiver': 'Samuel Carter', 'transaction_id': '51732411227'}
-----
Loaded 1691 real transactions from XML.

Total records for comparison: 1691

Performance comparison (200 lookups):
  Linear search:      0.021516 seconds
  Dictionary lookup: 0.000065 seconds
  → Dictionary is 329.5x faster
```

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

Linear search has $O(n)$ time complexity in the worst and average case. It must compare the target ID against every single record in the list.

Dictionary lookup has $O(1)$ 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:

Approach	Time Complexity	When to use it	Trade Offs
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)	$\sim O(\log n)$ 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)

Reflection on Basic Auth Limitations

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

1. Lack of Encryption (Encoding vs. Encryption)

Basic Auth credentials are sent as a Base64 string. **Base64 is not encryption**; 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.

2. Credential Persistence

With Basic Auth, the client (browser or Postman) must send the username and password with **every single request**. 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.

3. Vulnerability to Replay Attacks

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.

4. Production Alternatives

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

- **JWT (JSON Web Tokens):** Allows for short-lived sessions that expire automatically.
- **OAuth 2.0:** Separates authentication from authorization, common in banking APIs.
- **TLS/SSL (HTTPS):** Mandatory to encrypt the communication channel so that even Basic Auth headers cannot be read by third parties.