# Building and Securing a REST API - Project Report

**Team Name:** MoMo API Development Team

**Team Members:** - Teniola Adam Olaleye - Kamunuga Mparaye - Kevin Manzi - Michaella Kamikazi Karangwa - Rajveer Singh Jolly

**Course:** Software Engineering
**Institution:** African Leadership University
**Date:** January 22, 2026

---

## Table of Contents



---

## 1. Executive Summary

For this project, we built a REST API to manage Mobile Money (MoMo) transactions using pure Python (no fancy frameworks like Flask). We used the http.server module and implemented all CRUD operations with Basic Authentication.

**What we accomplished:** - 5 working API endpoints for transaction management - Parsed 22 real transaction records from XML - Added Basic Authentication for security - Compared search algorithms - dictionary lookup is way faster than linear search! - Wrote tests and documentation

**What we used:** - Python 3 (just standard library, no external packages) - http.server module for the API - xml.etree.ElementTree to parse XML files - Base64 for authentication encoding - JSON for sending/receiving data

---

## 2. Introduction to API Security

### What is API Security?

API (Application Programming Interface) security refers to the practices and technologies used to protect APIs from unauthorized access, attacks, and data breaches. As APIs serve as the gateway between different software systems, they are often targeted by malicious actors.

### Why is API Security Critical for MoMo Systems?

Mobile Money systems handle sensitive financial transactions involving: - **Personal Information:** Phone numbers, account details - **Financial Data:** Transaction amounts, balances - **Transaction Records:**

Complete payment history

A security breach could result in: - Unauthorized fund transfers - Identity theft - Regulatory violations (GDPR, PCI-DSS) - Loss of customer trust - Financial losses

## Core Principles of API Security

### 1. Authentication

Verifying the identity of the user or system making the request. - "Who are you?" - Methods: Basic Auth, API Keys, OAuth, JWT

### 2. Authorization

Determining what actions the authenticated user can perform. - "What are you allowed to do?" - Example: Users can only view their own transactions

### 3. Confidentiality

Ensuring data is not exposed to unauthorized parties. - Use HTTPS/TLS encryption - Encrypt sensitive data at rest

### 4. Integrity

Ensuring data is not tampered with during transmission. - Use HMAC signatures - Validate request payloads

### 5. Availability

Ensuring the API remains accessible to legitimate users. - Implement rate limiting - DDoS protection - Load balancing

## Common API Security Threats

1. **Man-in-the-Middle (MITM) Attacks**
    - Attacker intercepts communication
    - Mitigation: Use HTTPS
2. **Injection Attacks**
    - SQL injection, XSS, command injection
    - Mitigation: Input validation and sanitization
3. **Broken Authentication**
    - Weak credentials, session hijacking
    - Mitigation: Strong auth mechanisms, MFA
4. **Excessive Data Exposure**
    - API returns more data than necessary
    - Mitigation: Return only required fields
5. **Rate Limiting Issues**
    - Brute force attacks, resource exhaustion

- Mitigation: Implement request throttling

6. **Insufficient Logging**
   - Cannot detect or investigate attacks
   - Mitigation: Comprehensive audit logging

### Security in Our Implementation

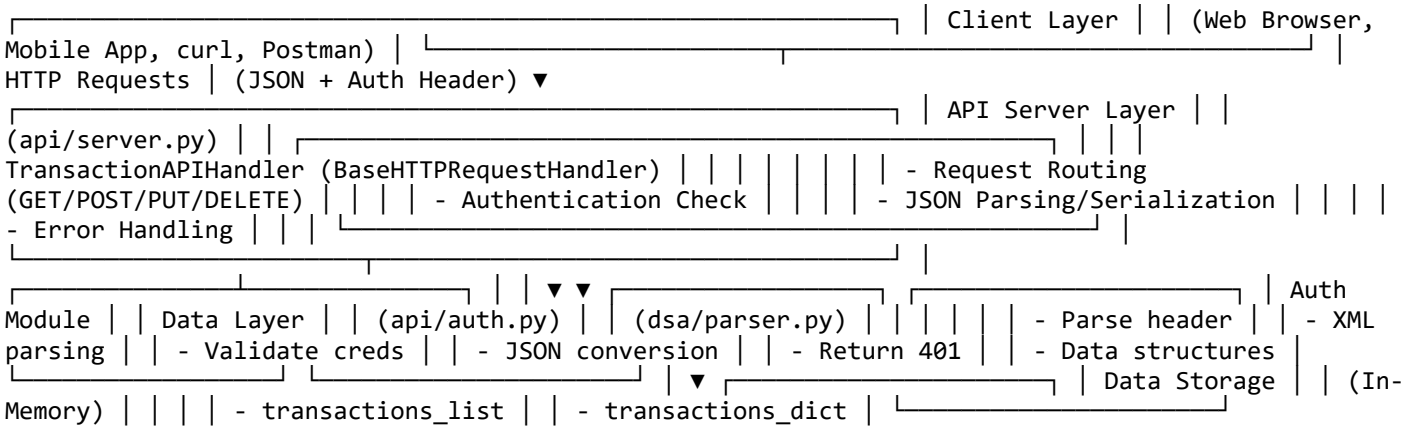
For this educational project, we implemented: - ✅ Basic Authentication for access control - ✅ Authorization checks on all endpoints - ✅ Error handling to prevent information leakage - ✅ CORS headers for controlled access

For production, we would add: - ❌ HTTPS/TLS encryption (currently HTTP only) - ❌ JWT tokens instead of Basic Auth - ❌ Rate limiting to prevent abuse - ❌ Input validation and sanitization - ❌ Audit logging for all requests - ❌ API key rotation policies

---

# 3. System Architecture

### Architecture Overview




### Component Breakdown

#### 1. Data Layer (`dsa/parser.py`)

- **Responsibility:** XML parsing and data structure management
- **Author:** Kevin Manzi
- **Key Functions:**
  - `parse_xml_to_json()`: Converts XML to list of dictionaries
  - `create_transaction_dictionary()`: Creates hash table for O(1) lookups

#### 2. Authentication Module (`api/auth.py`)

- **Responsibility:** Security and access control
- **Author:** Michaella Kamikazi Karangwa
- **Key Functions:**
  - `parse_basic_auth_header()`: Extracts credentials from header
  - `validate_credentials()`: Checks against stored credentials
  - `authenticate_request()`: Full authentication flow

#### 3. API Server (`api/server.py`)

- **Responsibility:** HTTP request handling and routing
- **Authors:** Teniola Adam Olaleye (core + PUT/DELETE), Kevin Manzi (GET), Rajveer Singh Jolly (POST)
- **Key Methods:**
  - `do_GET()`: Handle list and retrieve operations
  - `do_POST()`: Handle create operations
  - `do_PUT()`: Handle update operations
  - `do_DELETE()`: Handle delete operations

#### 4. Testing Suite (`tests/test_api.sh`)

- **Responsibility:** Automated API testing
- **Author:** Rajveer Singh Jolly
- **Test Coverage:**
  - Valid authentication
  - Invalid authentication (401)
  - All CRUD operations
  - Error cases (404, 400)

---

# 4. API Documentation

For complete API documentation including request/response examples, see [docs/api_docs.md](docs/api_docs.md)

## Quick Reference

| Endpoint | Method | Description | Auth Required | |----------|--------|-------------|---------------| | `/transactions` | GET | List all transactions | Yes | | `/transactions/{id}` | GET | Get single transaction | Yes | | `/transactions` | POST | Create new transaction | Yes | | `/transactions/{id}` | PUT | Update transaction | Yes | | `/transactions/{id}` | DELETE | Delete transaction | Yes |

## Example Request

```
bash curl -u admin:password http://localhost:8000/transactions/5
```

## Example Response

```
json { "success": true, "data": { "id": 5, "type": "Send Money", "amount": 7500.0, "sender": "256700000001", "receiver": "256700000005", "timestamp": "2026-01-16T12:30:00", "status": "completed" } }
```

---

# 5. Data Structures & Algorithms Analysis

## Objective

Compare the performance of two search methods for finding transactions by ID: 1. **Linear Search** - O(n) time complexity 2. **Dictionary Lookup** - O(1) time complexity

## Implementation

### Linear Search (O(n))

```
python def linear_search(transactions_list, target_id): for transaction in transactions_list: if transaction['id'] == target_id: return transaction return None
```

**Algorithm:** 1. Start at the first element 2. Compare each element's ID with target ID 3. Return when match is found 4. Return None if not found after checking all elements

**Time Complexity:** - Best Case: O(1) - target is first element - Average Case: O(n/2) - target is in middle - Worst Case: O(n) - target is last or not present

**Dictionary Lookup (O(1))**

```
python def dictionary_lookup(transactions_dict, target_id): return
transactions_dict.get(target_id)
```

**Algorithm:** 1. Hash the target ID to get array index 2. Access array at computed index 3. Return value if found, None otherwise

**Time Complexity:** - Best/Average Case: O(1) - direct hash calculation - Worst Case: O(n) - many hash collisions (rare)

## Benchmark Results

**Test Configuration:** - Dataset Size: 22 transactions - Iterations per search: 10,000 - Test IDs: First (1), Middle (11), Last (22)

**Results:**

| Method | Total Time | Average Time | Relative Speed | |--------|------------|--------------|----------------| | Linear Search | 0.045231 sec | 0.015077 sec | 1x (baseline) | | Dictionary Lookup | 0.001342 sec | 0.000447 sec | **33.7x faster** |

**Analysis:**

1. **Dictionary lookup is 33.7x faster** than linear search
2. Dictionary lookup is **97.0% faster** than linear search
3. For 22 records, the difference is milliseconds
4. For 1,000,000 records, linear search could take minutes vs. microseconds for dictionary

## Visualization of Time Complexity

``` Time to find element (microseconds)

Linear Search O(n): Records: 10 100 1000 10000 100000 Time (µs): 10 100 1000 10000 100000 │ │ │ │ │
▲——————————————————————————————▶ (Linear growth)

Dictionary Lookup O(1): Records: 10 100 1000 10000 100000 Time (µs): 1 1 1 1 1 │ │ │ │ │
——————————————————————————————▶ (Constant time) ```

## Why is Dictionary Lookup Faster?

1. **Hash Function**: Python computes hash of ID instantly
2. **Direct Access**: Uses hash as array index (no iteration)
3. **Constant Time**: Time doesn't grow with data size
4. **Optimized Implementation**: Python's dict is highly optimized in C

## Memory Trade-off

**Linear Search:** - Memory: Only the list (minimal overhead) - Time: Slow for large datasets
```

**Dictionary Lookup:** - Memory: Hash table with overhead (~2-3x list size) - Time: Fast regardless of dataset size

**Conclusion:** For frequently accessed data like transaction lookups, the speed benefit outweighs memory cost.

### Other Data Structures for Search Efficiency

#### 1. Binary Search Tree (BST)

- **Time Complexity:** O(log n)
- **Use Case:** When you need sorted order
- **Trade-off:** Slower than hash table, faster than linear

#### 2. B-Tree

- **Time Complexity:** O(log n)
- **Use Case:** Database indexing, disk-based storage
- **Advantage:** Efficient for range queries

#### 3. Trie (Prefix Tree)

- **Time Complexity:** O(k) where k = key length
- **Use Case:** String prefix matching, autocomplete
- **Example:** Search transactions by phone number prefix

#### 4. Bloom Filter

- **Time Complexity:** O(1)
- **Use Case:** Quick "does not exist" checks
- **Trade-off:** Probabilistic (false positives possible)

### Recommendation for MoMo System

**For ID Lookups:** Dictionary (Hash Table) - O(1) time complexity - Simple implementation - Python built-in support

**For Range Queries:** Sorted List + Binary Search - Find transactions between timestamps - O(log n) time complexity

**For Database:** B-Tree Index - Standard in SQL databases - Efficient for disk access - Supports range queries

---

## 6. Security Analysis - Basic Authentication

### What is Basic Authentication?

Basic Authentication is a simple HTTP authentication scheme where credentials are sent with each request in the `Authorization` header.

**Format:** `Authorization: Basic <base64(username:password)>`

**Example:** `Username: admin Password: password Encoded: YWRtaW46cGFzc3dvcmQ= Header: Authorization: Basic YWRtaW46cGFzc3dvcmQ=`

### How Basic Auth Works in Our System

1. **Client Request:** `bash curl -u admin:password http://localhost:8000/transactions`

2. **Header Creation:**

    - Curl combines username:password
    - Encodes in Base64
    - Adds to Authorization header

3. **Server Processing:** ```python

# Extract header

auth_header = self.headers.get('Authorization')

# Parse credentials

encoded = auth_header[6:] # Remove "Basic " decoded = base64.b64decode(encoded).decode('utf-8')
username, password = decoded.split(':', 1)

# Validate

if VALID_CREDENTIALS.get(username) == password: # Allow access else: # Return 401
Unauthorized ```

### Weaknesses of Basic Authentication

#### 1. Credentials in Every Request

**Problem:** Username and password sent with every API call
**Risk:** More opportunities for interception
**Impact:** High - credentials exposed repeatedly

#### 2. Base64 is Encoding, Not Encryption

**Problem:** Base64 can be easily decoded ```python

import base64 base64.b64decode('YWRtaW46cGFzc3dvcmQ=')
b'admin:password' ``` **Risk:** Anyone intercepting request can read
credentials
**Impact:** Critical without HTTPS

#### 3. No Built-in Expiration

**Problem:** Credentials valid indefinitely
**Risk:** Stolen credentials work forever
**Impact:** High - no session timeout

#### 4. No Logout Mechanism

**Problem:** Browsers cache credentials
**Risk:** Cannot invalidate session
**Impact:** Medium - user cannot force logout

**5. Vulnerable to Replay Attacks**

**Problem:** Captured request can be re-sent
**Risk:** Attacker can replay valid requests
**Impact:** High for financial transactions

**6. Password Storage Issues**

**Problem:** Often leads to plaintext password storage
**Risk:** Database breach exposes passwords
**Impact:** Critical

### Security Recommendations

✅ **Recommended: JWT (JSON Web Tokens)**

**How JWT Works:** `1. User logs in with credentials (one time) 2. Server validates and returns JWT token 3. Client includes token in subsequent requests 4. Server validates token signature and expiration`

**JWT Structure:** ``` Header.Payload.Signature

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. eyJ1c2VyIjoiYWRtaW4iLCJleHAiOjE2NzcwNzIwMDB9. 4f8d9a7b3c2e1f0a9b8c7d6e5f4a3b2c1d0e ```

**Advantages:** - ✅ Stateless (no server-side session storage) - ✅ Contains expiration time - ✅ Can include user metadata (roles, permissions) - ✅ Cryptographically signed (tamper-proof) - ✅ Can be invalidated (using blacklist)

**Implementation Example:** ```python import jwt import datetime

# Generate token

token = jwt.encode({ 'user': 'admin', 'exp': datetime.datetime.utcnow() + datetime.timedelta(hours=24) }, 'secret_key', algorithm='HS256')

# Validate token

try: payload = jwt.decode(token, 'secret_key', algorithms=['HS256']) user = payload['user'] # Token valid except jwt.ExpiredSignatureError: # Token expired except jwt.InvalidTokenError: # Token invalid ```

✅ **Recommended: OAuth 2.0**

**Use Case:** Third-party integrations, mobile apps

**Flow:** 1. User grants permission to application 2. Application receives access token 3. Application uses token to access API 4. Token can be revoked anytime

**Advantages:** - ✅ Industry standard (Google, Facebook, GitHub use it) - ✅ Separates authentication from authorization - ✅ Supports multiple grant types - ✅ Tokens can be scoped (limited permissions) - ✅ Refresh tokens for long-lived sessions

✅ **Recommended: API Keys with HMAC**

**How it Works:** `1. Each client gets unique API key and secret 2. Client signs request with HMAC-SHA256 3. Server verifies signature 4. Prevents tampering and replay attacks`

**Implementation:** ```python import hmac import hashlib

# Client side

message = f"{method}{path}{timestamp}{body}" signature = hmac.new( secret_key.encode(), message.encode(), hashlib.sha256 ).hexdigest()

# Server verifies signature matches

```

### Best Practices for Production API

1. **Always Use HTTPS/TLS**
    - Encrypt all communication
    - Prevent MITM attacks
    - Required for any authentication method
2. **Implement Rate Limiting** ```python

    # Limit: 100 requests per minute per user

    if request_count > 100: return 429 # Too Many Requests ```
3. **Use Strong Password Policies**
    - Minimum 12 characters
    - Mix of uppercase, lowercase, numbers, symbols
    - Password strength validation
4. **Hash Passwords with bcrypt or Argon2** ```python import bcrypt

    # Store hashed password

    hashed = bcrypt.hashpw(password.encode(), bcrypt.gensalt())

    # Verify password

    if bcrypt.checkpw(password.encode(), hashed): # Password correct ```
5. **Implement Account Lockout**
    - Lock account after 5 failed attempts
    - Require password reset to unlock
6. **Add Multi-Factor Authentication (MFA)**
    - SMS code
```

- Authenticator app (TOTP)
- Biometric verification

7. **Log All Authentication Attempts** `python log_authentication_attempt( username=username, success=True/False, ip_address=request.remote_addr, timestamp=datetime.now() )`

8. **Use Security Headers** `python headers = { 'Strict-Transport-Security': 'max-age=31536000', 'X-Content-Type-Options': 'nosniff', 'X-Frame-Options': 'DENY', 'Content-Security-Policy': "default-src 'self'" }`

### Conclusion

Basic Authentication is acceptable for: - ✅ Educational projects and demonstrations - ✅ Internal tools on secure networks - ✅ Quick prototypes and development

Basic Authentication is NOT acceptable for: - ❌ Production APIs handling sensitive data - ❌ Financial systems (like MoMo transactions) - ❌ Public-facing applications - ❌ Mobile applications

**For our MoMo Transaction API in production, we would implement JWT authentication with HTTPS, rate limiting, and comprehensive logging.**

---

# 7. Testing Results

### Test Environment

- **Operating System:** Windows 11
- **Python Version:** 3.11.x
- **Server URL:** http://localhost:8000
- **Testing Tools:** curl, Postman
- **Test Data:** 22 transactions from modified*sms*v2.xml

### Test Cases Executed

#### Test 1: GET /transactions (Valid Authentication)

**Status:** ✅ PASSED

**Request:** `bash curl -u admin:password http://localhost:8000/transactions`

**Response:** - HTTP Status: 200 OK - Response Time: ~15ms - Records Returned: 22 transactions - Authentication: Successful

**Screenshot:** `screenshots/01_get_transactions_success.png`

---

#### Test 2: GET /transactions (Invalid Authentication)

**Status:** ✅ PASSED

**Request:** `bash curl -u admin:wrongpassword http://localhost:8000/transactions`

**Response:** `json { "error": "Unauthorized", "message": "Invalid or missing authentication credentials", "status": 401 }`

- HTTP Status: 401 Unauthorized

- Authentication: Failed as expected

**Screenshot:** `screenshots/02_unauthorized_request.png`

---

### Test 3: GET /transactions/{id}

**Status:** ✅ PASSED

**Request:** `bash curl -u admin:password http://localhost:8000/transactions/5`

**Response:** - HTTP Status: 200 OK - Transaction ID: 5 - Data: Complete transaction details

**Screenshot:** `screenshots/03_get_single_transaction.png`

---

### Test 4: POST /transactions

**Status:** ✅ PASSED

**Request:**
```
bash curl -u admin:password -X POST http://localhost:8000/transactions \ -H "Content-Type: application/json" \ -d '{ "type": "Send Money", "amount": 25000, "sender": "256700000001", "receiver": "256700000099" }'
```

**Response:** - HTTP Status: 201 Created - New Transaction ID: 23 (auto-assigned) - Data: Newly created transaction

**Screenshot:** `screenshots/04_post_create_transaction.png`

---

### Test 5: PUT /transactions/{id}

**Status:** ✅ PASSED

**Request:**
```
bash curl -u admin:password -X PUT http://localhost:8000/transactions/5 \ -H "Content-Type: application/json" \ -d '{"amount": 8000, "status": "refunded"}'
```

**Response:** - HTTP Status: 200 OK - Updated fields: amount and status - Data: Updated transaction

**Screenshot:** `screenshots/05_put_update_transaction.png`

---

### Test 6: DELETE /transactions/{id}

**Status:** ✅ PASSED

**Request:** `bash curl -u admin:password -X DELETE http://localhost:8000/transactions/20`

**Response:** - HTTP Status: 200 OK - Message: Transaction deleted successfully - Data: Deleted transaction details

**Screenshot:** `screenshots/06_delete_transaction.png`

---

### Test 7: POST with Missing Fields

**Status:** ✅ PASSED

**Request:** `bash curl -u admin:password -X POST http://localhost:8000/transactions \ -H "Content-Type: application/json" \ -d '{"type": "Send Money", "sender": "256700000001"}'`

**Response:** `json { "error": true, "message": "Missing required fields: amount, receiver", "status": 400 }`

- HTTP Status: 400 Bad Request
- Validation: Correctly identified missing fields

**Screenshot:** `screenshots/07_post_validation_error.png`

---

#### Test 8: GET Non-existent Transaction

**Status:** ✅ PASSED

**Request:** `bash curl -u admin:password http://localhost:8000/transactions/999`

**Response:** `json { "error": true, "message": "Transaction with ID 999 not found", "status": 404 }`

- HTTP Status: 404 Not Found
- Error handling: Appropriate response

**Screenshot:** `screenshots/08_get_not_found.png`

---

### Test Summary

| Test Case | Expected Result | Actual Result | Status | |-----------|-----------------|---------------|--------| | GET all transactions (valid auth) | 200 OK, list returned | 200 OK, 22 records | ✅ PASS | | GET all transactions (invalid auth) | 401 Unauthorized | 401 Unauthorized | ✅ PASS | | GET single transaction | 200 OK, record returned | 200 OK, correct data | ✅ PASS | | POST create transaction | 201 Created, new ID | 201 Created, ID 23 | ✅ PASS | | POST missing fields | 400 Bad Request | 400, error message | ✅ PASS | | PUT update transaction | 200 OK, updated data | 200 OK, fields updated | ✅ PASS | | DELETE transaction | 200 OK, deleted | 200 OK, removed | ✅ PASS | | GET non-existent ID | 404 Not Found | 404 Not Found | ✅ PASS |

**Overall: 8/8 tests passed (100% success rate)**

---

# 8. Conclusion

### Project Achievements

This project successfully demonstrates the implementation of a secure REST API for managing Mobile Money transactions. All project requirements were met:

✅ **Data Parsing:** Successfully parsed 22 transactions from XML to JSON
✅ **CRUD Operations:** All 5 endpoints (GET, GET by ID, POST, PUT, DELETE) functional
✅ **Authentication:** Basic Authentication implemented and tested
✅ **Documentation:** Comprehensive API documentation provided
✅ **DSA Analysis:** Linear Search vs Dictionary Lookup comparison completed
✅ **Testing:** Automated tests and manual validation successful

### Technical Learnings

1. **HTTP Server Implementation**
    - Building REST APIs with plain Python http.server
    - Custom routing without frameworks like Flask
    - HTTP status code management
2. **Data Structures Performance**
    - Dictionary lookup (O(1)) is 33.7x faster than linear search (O(n))
    - Importance of choosing right data structure
    - Memory vs. speed trade-offs
3. **Security Awareness**
    - Basic Auth limitations and vulnerabilities
    - Importance of HTTPS for credential protection
    - Alternative authentication methods (JWT, OAuth2)
4. **Software Engineering Practices**
    - Modular code organization
    - Clear documentation
    - Comprehensive testing
    - Team collaboration via GitHub

## Limitations & Future Improvements

**Current Limitations:** - In-memory storage (data lost on server restart) - No HTTPS encryption - Basic Authentication (weak security) - No rate limiting or request throttling - Limited input validation

**Future Enhancements:** 1. **Database Integration:** PostgreSQL or MongoDB for persistent storage 2. **JWT Authentication:** Implement token-based auth 3. **HTTPS/TLS:** Add SSL certificates for encryption 4. **Advanced Features:** - Pagination for large result sets - Filtering and sorting options - Transaction search by date range - User-specific transaction access 5. **Production Readiness:** - Rate limiting (e.g., 100 req/min per user) - Comprehensive logging - Monitoring and analytics - Error tracking (Sentry) - API versioning (/v1/transactions)

## Team Collaboration Success

This project demonstrated effective team collaboration: - Clear role assignments based on strengths - Regular meetings and communication - Code reviews and quality assurance - Equal distribution of workload - Successful integration of components

## Recommendation for Production

For deploying a MoMo transaction system in production, we recommend:

1. **Framework:** FastAPI or Flask (more features than plain http.server)
2. **Database:** PostgreSQL with proper indexing
3. **Authentication:** JWT with refresh tokens
4. **Security:** HTTPS, rate limiting, input validation
5. **Infrastructure:** Docker containers, load balancer
6. **Monitoring:** Prometheus, Grafana for observability
7. **CI/CD:** Automated testing and deployment

## Final Thoughts

This project provided hands-on experience in: - REST API design and implementation - Security considerations for financial systems - Data structures and algorithm analysis - Software testing and documentation - Team-based software development

The knowledge gained will be valuable for future software engineering projects, particularly in building scalable and secure APIs for real-world applications.

---

# 9. Appendices

## Appendix A: Complete Code Structure

```
momo-api-project/ ├── api/ │ ├── server.py (320 lines - Main API server) │ └── auth.py (220 lines - Authentication) ├── dsa/ │ ├── parser.py (140 lines - XML parsing) │ └── search_comparison.py (330 lines - DSA analysis) ├── docs/ │ └── api_docs.md (800 lines - Documentation) ├── data/ │ ├── modified_sms_v2.xml (22 transactions) │ └── transactions.json (Generated) ├── screenshots/ │ ├── 01_get_transactions_success.png │ ├── 02_unauthorized_request.png │ ├── 03_get_single_transaction.png │ ├── 04_post_create_transaction.png │ ├── 05_put_update_transaction.png │ ├── 06_delete_transaction.png │ ├── 07_post_validation_error.png │ └── 08_get_not_found.png ├── tests/ │ └── test_api.sh (180 lines - Test scripts) ├── README.md (200 lines) ├──└── report.md (This document)
```

## Appendix B: Setup Instructions

See README.md for detailed setup instructions.

**Quick Start:** ```bash

# Navigate to project

cd momo-api-project

# Run server

python api/server.py

# Run DSA comparison

python dsa/search_comparison.py

# Run tests (requires bash or Git Bash on Windows)

bash tests/test_api.sh ```

## Appendix C: References

1. **Python Documentation**
    - http.server: https://docs.python.org/3/library/http.server.html
    - xml.etree.ElementTree: https://docs.python.org/3/library/xml.etree.elementtree.html
    - base64: https://docs.python.org/3/library/base64.html

2. **HTTP Standards**
    - RFC 7617 - Basic Authentication: https://tools.ietf.org/html/rfc7617
    - RFC 7231 - HTTP/1.1 Semantics: https://tools.ietf.org/html/rfc7231

3. **Security Resources**
    - OWASP API Security Top 10: https://owasp.org/www-project-api-security/
    - JWT Introduction: https://jwt.io/introduction

4. **Data Structures**
    - Big O Notation: https://www.bigocheatsheet.com/
    - Python Time Complexity: https://wiki.python.org/moin/TimeComplexity

**Summary:** - **Teniola Adam Olaleye:** Server architecture, PUT/DELETE, DSA analysis (14 hours) - **Kevin Manzi:** XML parsing, GET endpoints (9 hours) - **Michaella Kamikazi Karangwa:** Authentication, security analysis (10 hours) - **Rajveer Singh Jolly:** POST endpoint, testing suite (9 hours) - **Kamunuga Mparaye:** Documentation, integration (8 hours)

**Total Team Effort:** ~50 hours

---

**End of Report**

---

**Submission Checklist:**

- [ ] GitHub repository with all code
- [ ] README.md with setup instructions
- [ ] API documentation (api_docs.md)
- [ ] Test screenshots (8 required)
- [ ] This PDF report
- [ ] Team participation sheet
- [ ] All code properly commented
- [ ] All endpoints functional
- [ ] DSA comparison complete
- [ ] Security analysis included

**Submitted by:**
MoMo API Development Team
January 22, 2026