CampusMint Technical Implementation Guide

Table of Contents

System Architecture

Overview

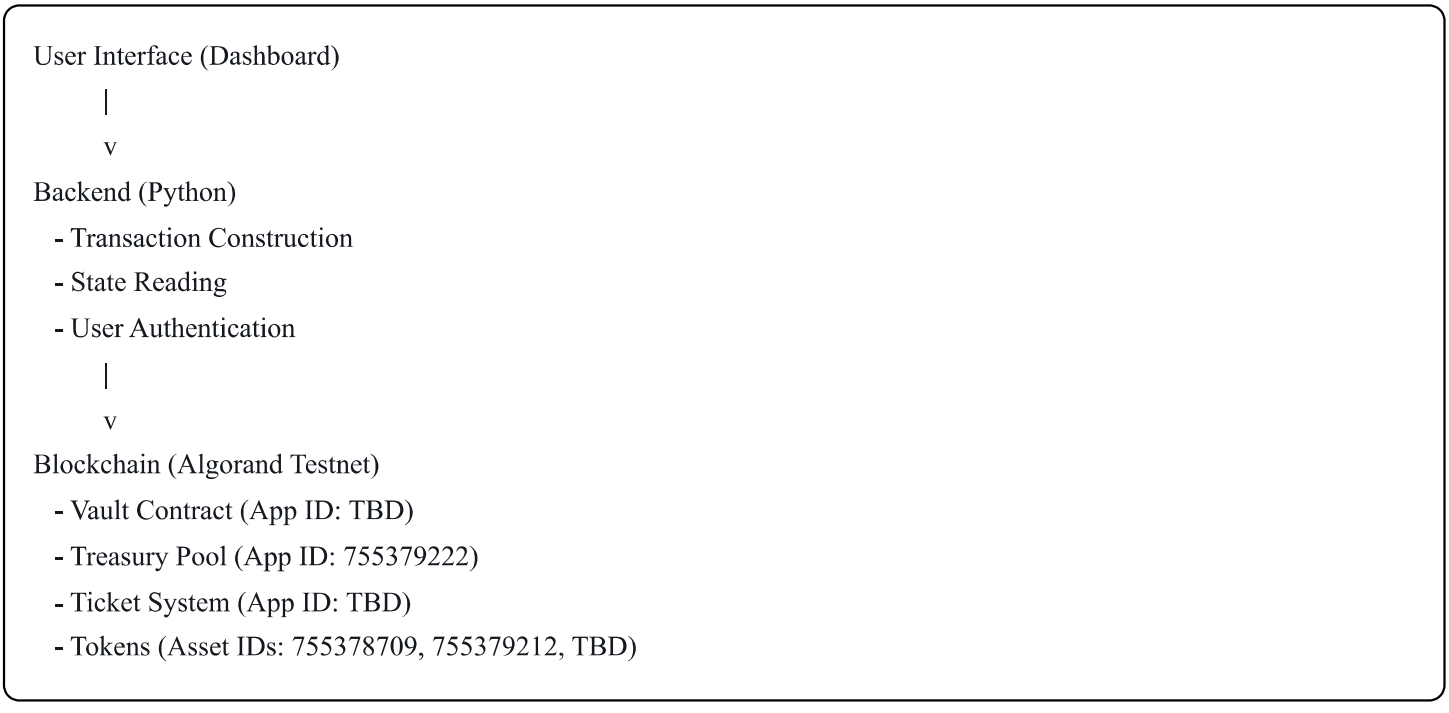
CampusMint consists of three main layers:

Blockchain Layer: Smart contracts running on Algorand testnet. Manages all transaction logic and state.

Backend Layer: Python backend that connects to the blockchain, reads state, and constructs transactions.

Frontend Layer: Dashboard interface that students use to interact with the system.

Architecture Diagram



Technology Stack

Blockchain: Algorand testnet
Smart Contracts: TEAL 6.0
Backend: Python 3.8+
Libraries: py-algorand-sdk, FastAPI or Flask
Frontend: Jupyter notebook / Google Colab
Database: None (all state on blockchain)

Smart Contracts

Contract 1: Vault (Student Savings)

Purpose: Enforce time-locked savings. Student deposits money, smart contract locks it until a specified date.

Global State Variables

owner (bytes): Address that created the vault
beneficiary (bytes): Address that can withdraw funds
locked_amount (uint64): Amount locked (in microunits, 1/100th of token unit)
unlock_time (uint64): Unix timestamp when funds can be withdrawn
token_asset_id (uint64): Which ASA token is being locked
purpose (bytes): Description of why funds are being saved

Contract Methods

Create Vault

Input:
- beneficiary_address: Address that can withdraw
- amount: How much to lock (in token units)
- unlock_timestamp: When the vault unlocks
- purpose: String describing the savings goal

Process:
1. Validate: amount > 0, unlock_time > current_time
2. Validate: beneficiary address is valid
3. Validate: token_asset_id is valid
4. Transfer tokens from owner to contract
5. Store all parameters in global state
6. Return: Vault created successfully

Security:
- Only owner can create vault for themselves
- Unlock time must be in the future
- Contract holds the tokens, not the application

Check Status

Input: None (query operation)

Output:
- locked_amount
- unlock_time
- is_unlocked (boolean: current_time >= unlock_time)
- owner
- beneficiary

Process:
1. Read global state
2. Compare current time to unlock_time
3. Return all information

Withdraw

Input: None (authenticated operation)

Process:

1. Verify: sender == beneficiary
2. Verify: current_time >= unlock_time
3. Transfer locked_amount to beneficiary
4. Clear global state
5. Return: Withdrawal successful

Security:

- Only beneficiary can withdraw
- Only after unlock time
- Funds are held by contract, cannot be stolen

Deployment

The vault contract is deployed once per campus or organization. Each deployment creates a new app ID. Deployment includes:

1. Compile TEAL code to bytecode
2. Deploy to Algorand testnet
3. Initialize contract state
4. Record app ID

Once deployed, students interact with this single instance. Multiple vaults (one per student) are stored in the contract's local state.

Testing

Local testing uses Algorand sandbox. Tests verify:

1. Vaults can be created with valid parameters
2. Vaults cannot be created with invalid parameters
3. Beneficiary can withdraw after unlock time
4. Beneficiary cannot withdraw before unlock time
5. Owner cannot withdraw early
6. Non-beneficiary cannot withdraw
7. Vault state updates correctly

Contract 2: Treasury Pool (Club Fundraising)

Purpose: Manage fundraising campaigns. Club sets a goal and deadline. Students contribute. Funds are released when goal is met.

Global State Variables

admin (bytes): Address managing the fundraiser
goal (uint64): Fundraising target in token units
total_raised (uint64): Amount contributed so far
deadline (uint64): Unix timestamp when fundraiser ends
token_asset_id (uint64): Which token is accepted (usually CampusCoin)
beneficiary (bytes): Address that receives funds if goal is met
purpose (bytes): Description of what funds will be used for

Contract Methods

Create Treasury Pool

Input:
- admin_address
- goal_amount
- deadline_timestamp
- beneficiary_address
- purpose_string

Process:
1. Validate all parameters
2. Store in global state
3. Initialize total_raised to 0
4. Return: Treasury created

Security:
- Only admin can create
- Goal must be > 0
- Deadline must be in future

Contribute

Input:
- amount: How much to contribute

Process:
1. Verify: current_time < deadline (fundraiser still active)
2. Receive tokens from contributor
3. Update total_raised += amount
4. Record contributor address and amount
5. Return: Contribution recorded

Security:
- Cannot contribute after deadline
- Amount must be > 0
- Tokens are held by contract

Claim Funds

Input: None (authenticated to admin)

Process:
1. Verify: sender == admin
2. Verify: current_time >= deadline OR total_raised >= goal
3. If goal was met: transfer all funds to beneficiary
4. If goal was not met: keep funds for refund period
5. Return: Funds claimed or refund available

Security:
- Only admin can claim
- Only after deadline
- Only if goal was met

Get Refund

Input: None (query operation)

Process:
1. Verify: goal was not met
2. Verify: contributor has contributed
3. Calculate refund amount
4. Transfer refund to contributor
5. Clear record of contribution
6. Return: Refund processed

Security:
- Only if goal failed
- Only for those who contributed
- Only after deadline

Deployment

A treasury pool is deployed when a club wants to run a fundraiser. Each fundraiser is a separate contract instance. Deployment includes:

1. Compile TEAL code
2. Deploy to Algorand testnet
3. Initialize with goal, deadline, beneficiary
4. Share app ID with students

The treasury pool at app ID 755379222 is a test instance currently running.

Testing

Tests verify:

1. Pool can be created with valid parameters
2. Pool cannot be created with invalid parameters
3. Multiple contributions can be made
4. Total raised updates correctly
5. Cannot contribute after deadline
6. Admin can claim funds when goal is met
7. Admin cannot claim funds when goal is not met
8. Contributors can request refunds if goal fails
9. Progress calculation is accurate

Contract 3: NFT Ticket System

Purpose: Issue unique tickets as NFTs. Students purchase tickets. Tickets cannot be duplicated.

Implementation Approach

The ticket system uses Algorand's standard ASA (Algorand Standard Asset) with supply = 1.

For each ticket:

1. Club creates an ASA with total supply of 1
2. ASA has metadata describing the event
3. Student purchases by sending payment
4. Payment is monitored by backend
5. Upon confirmation, ASA is transferred to student
6. Student can show the ASA as proof of ownership

This approach is simpler than a custom smart contract. It leverages Algorand's native asset functionality.

Asset Properties

Name: Event name + seat/ticket number (example: "AI Fest Seat 1")
Unit Name: "TICKET"
Total Supply: 1
Decimals: 0 (indivisible)
Manager: Club organization address
Reserve: Club address
Freeze: Manager (prevents unauthorized trading)
Clawback: Disabled (prevents admin reclaiming)

Purchase Flow

1. Club mints ASAs (one per ticket)
2. Club publishes list of available tickets
3. Student sees ticket in dashboard
4. Student clicks "Buy"
5. Student sends payment in Campus INR
6. Backend monitors blockchain for payment
7. Upon confirmation, backend transfers ASA to student
8. Backend confirms transaction ID to student

Verification Flow

1. Event day arrives
2. Organizer has list of ticket ASA IDs
3. Student arrives at event
4. Organizer scans student's address (via QR code or manual entry)
5. Organizer checks blockchain: does student own the ASA?
6. If yes: student enters
7. If no: student is denied entry

Testing

Tests verify:

1. ASAs can be created with correct properties
2. ASA supply is exactly 1
3. ASA transfers to buyer upon payment
4. ASA cannot be duplicated
5. Ownership can be verified on blockchain
6. Transfers can be frozen if needed

Dashboard Application

Architecture

The dashboard is a Python application running in Google Colab. It provides a text-based interface to the blockchain.

Core Components

Blockchain Reader: Connects to Algorand testnet, reads contract state and account balances.

Transaction Constructor: Builds transactions when students take actions.

User Interface: Displays information and accepts user input.

Configuration: Manages app IDs, asset IDs, and other parameters.

Blockchain Reader

The reader module provides functions to query blockchain state without modifying it.

```python
def get_asset_info(asset_id):
    """Retrieve asset metadata from blockchain"""
    # Connect to Algorand
    # Query asset parameters
    # Return: name, supply, decimals, creator, etc.

def get_account_balance(address, asset_id=None):
    """Get account balance for a specific token"""
    # If asset_id is None: return ALGO balance
    # If asset_id is provided: return token balance
    # Return: balance in human-readable format

def read_app_global_state(app_id):
    """Read smart contract global state"""
    # Connect to contract
    # Decode base64-encoded state
    # Return: dictionary of state variables

def read_vault_state(app_id):
    """Parse vault contract state"""
    # Read global state
    # Decode lock time to human-readable date
    # Check if vault is unlocked
    # Return: formatted vault information

def read_treasury_state(app_id):
    """Parse treasury contract state"""
    # Read global state
    # Calculate progress percentage
    # Check if deadline has passed
    # Return: formatted treasury information
```

Transaction Constructor

This module builds transactions for user actions.

```python
```

```
def create_vault_transaction(owner, beneficiary, amount, unlock_time, token_id):
    """Build a vault creation transaction"""
    # Construct transaction
    # Add app call arguments
    # Return: unsigned transaction

def create_contribution_transaction(contributor, treasury_app_id, amount):
    """Build a contribution transaction"""
    # Construct asset transfer to treasury
    # Return: unsigned transaction

def withdraw_vault_transaction(beneficiary, vault_app_id):
    """Build a vault withdrawal transaction"""
    # Construct app call
    # Return: unsigned transaction
```

User Interface

The dashboard provides a menu-driven interface.

Main Menu

```
1. View Account
   - Show ALGO balance
   - Show token balances
   - Show active vaults
   - Show ticket ownership

2. Vault Operations
   - Create vault
   - View vault status
   - Withdraw from vault

3. Fundraising
   - View active campaigns
   - Contribute to campaign
   - Request refund

4. Tickets
   - View available tickets
   - Purchase ticket
   - View owned tickets

5. Export Data
   - Export account state
   - Export transaction history

0. Exit
```

Each menu option guides the user through the necessary inputs and displays the result.

Configuration

The dashboard reads configuration from a JSON file:

```
json
```

```
{
 "ALGOD_ADDRESS": "https://testnet-api.algonode.cloud",
 "ALGOD_TOKEN": "",
 "NETWORK": "testnet",

 "VAULT": {
  "app_id": null,
  "token_asset_id": null
 },

 "TICKETS": {
  "payment_token_id": 755378709
 },

 "TREASURY": {
  "app_id": 755379222,
  "token_asset_id": 755379212
 }
}
```


When new contracts are deployed, the configuration is updated with the new app IDs and asset IDs.

Backend Integration

Backend Responsibilities

The backend handles interaction with the blockchain. It has three main responsibilities:

1. State Reading

The backend continuously monitors the blockchain for new blocks. When called, it reads smart contract state, account balances, and transaction history. This information is cached locally for performance.

2. Transaction Construction

When a user takes an action, the backend constructs the appropriate transaction. For example:

- Vault creation: Call vault contract with parameters
- Contribution: Transfer tokens to treasury contract
- Withdrawal: Call vault contract withdraw method

The transaction is returned to the user unsigned. The user signs it with their own key.

3. Transaction Broadcast

The signed transaction is sent to the blockchain. The backend monitors the blockchain to confirm the transaction was included in a block. Once confirmed, the user is notified.

API Endpoints

The backend provides REST API endpoints:

```
GET /api/account/{address}
  Returns: account balance and asset holdings

GET /api/vault/{app_id}
  Returns: vault state and unlock status

GET /api/treasury/{app_id}
  Returns: treasury state and progress

POST /api/transaction
  Input: unsigned transaction
  Returns: transaction ID after broadcast

GET /api/transaction/{txid}
  Returns: confirmation status
```

Error Handling

The backend handles errors gracefully:

Invalid Address: Returns 400 Bad Request
App Not Found: Returns 404 Not Found
Transaction Failed: Returns 400 Bad Request with reason
Network Error: Retries and returns 503 Service Unavailable

Deployment Guide

Prerequisites

Python 3.8 or higher installed Algorand SDK installed: `pip install py-algorand-sdk` Access to Algorand testnet faucet GitHub repository cloned locally

Step 1: Deploy Vault Contract

```
bash
```

```
# Compile TEAL to bytecode
goal app compile contracts/vault.teal

# Deploy to testnet
python deploy_vault.py

# Output: App ID (example: 123456789)

# Update configuration
# Set VAULT.app_id = 123456789
# Set VAULT.token_asset_id = [newly created asset ID]
```

Step 2: Deploy Ticket System

```
bash

# For each ticket, create an ASA
goal asset create --name "Event Seat 1" --total 1

# Record asset IDs
# Update configuration: TICKETS.ticket_asset_ids = [list of IDs]
```

Step 3: Deploy Dashboard

```
bash

# Upload to Google Colab
# Or run locally: jupyter notebook

# Configure with app IDs and asset IDs
# Run all cells

# Dashboard is now live
```

Step 4: Initialize Tokens

Before users can transact, accounts must opt-in to assets:

```
python
```

```
# For each token
# Each user must opt-in first
# Dashboard can guide users through this

# For testnet testing, we can pre-opt-in with admin account
```

Testing Deployment

```bash
# Test vault creation
python test_vault_creation.py

# Test contribution
python test_treasury_contribution.py

# Test ticket transfer
python test_ticket_transfer.py

# All tests pass: deployment is successful
```

Testing Strategy

Unit Testing

Test each contract function independently:

```python
```

```
# vault_test.py

def test_vault_creation():
    # Create vault with valid parameters
    # Assert vault state is set correctly
    # Assert tokens are transferred to contract

def test_withdraw_before_unlock():
    # Create vault
    # Try to withdraw before unlock time
    # Assert transaction fails

def test_withdraw_after_unlock():
    # Create vault
    # Wait until after unlock time
    # Withdraw
    # Assert funds transferred to beneficiary
```

Integration Testing

Test complete user flows:

```
python

def test_student_creates_vault():
    # Student creates vault
    # Dashboard shows vault
    # Progress toward unlock date is visible
    # Time passes
    # Student withdraws
    # Transaction succeeds

def test_club_runs_fundraiser():
    # Club creates treasury pool
    # Club shares with students
    # Student 1 contributes
    # Student 2 contributes
    # Progress updates
    # Deadline passes
    # Goal was met
    # Admin claims funds
    # Funds transferred to beneficiary
```

End-to-End Testing

Test the complete system with real user interactions:

```python
def test_production_scenario():
    # Real student creates account
    # Student gets testnet ALGO
    # Student opts into tokens
    # Student creates vault
    # Student views vault status
    # Time passes
    # Student withdraws
    # All steps succeed
```

Performance Testing

Test system performance under load:

```python
def test_concurrent_contributions():
    # 100 students contribute simultaneously
    # All transactions succeed
    # Treasury state is consistent
    # No data loss

def test_large_fundraiser():
    # Treasury pool raises 1 million rupees
    # Dashboard displays correctly
    # Transactions don't timeout
```

Monitoring and Maintenance

Monitoring

After deployment, the system should be monitored for:

- Contract state consistency
- Transaction success rates
- Error frequencies

- User adoption
- Performance metrics

Maintenance

Regular maintenance tasks:

- Update smart contracts if bugs are found
- Refresh tokens when supply approaches limit
- Archive completed fundraisers
- Monitor testnet for deprecation notices

Security Checklist

Before production deployment:

- [ ] All smart contracts audited by third party
- [ ] Private keys never logged or transmitted
- [ ] HTTPS enforced for all connections
- [ ] Rate limiting on API endpoints
- [ ] Input validation on all user inputs
- [ ] Error messages don't leak sensitive information
- [ ] Recovery procedures documented
- [ ] Incident response plan created

Conclusion

This technical guide provides the implementation details needed to build and deploy CampusMint. The system is modular, allowing independent development and testing of each component. Start with smart contracts, then build the dashboard, then integrate with the backend.

The architecture is designed for maintainability and scalability. New features can be added without modifying existing code. New instances can be deployed to other campuses using the same codebase.