Sui Move Hackathon - Easy Challenges

Week 3 Puzzles

Challenge 1: Counter with Access Control

Problem Description

Build a shared counter that anyone can increment, but only the owner can reset. This teaches you about shared objects, capabilities, and access control in Sui Move.

What You Need to Build

Core Functionality:

1. **Counter Creation**
    - Create a shared counter object
    - Initialize with starting value (default 0)
    - Store owner's address
    - Issue an AdminCap to the creator
2. **Increment Function**
    - Anyone can increment the counter
    - Increase counter by 1 or by a specified amount
    - Emit event when counter is incremented
3. **Get Value Function**
    - Public view function to read current counter value
    - No transaction needed (read-only)
4. **Reset Function**
    - Only admin (with AdminCap) can reset
    - Reset counter back to 0 or specified value
    - Emit event when counter is reset

Expected Behavior

```
None
Day 1:
- Alice creates counter → starts at 0
```

```
- Alice receives AdminCap

Day 2:
- Bob increments counter → value becomes 1
- Charlie increments counter by 5 → value becomes 6
- Anyone can check value → returns 6

Day 3:
- Bob tries to reset counter → FAILS (no AdminCap)
- Alice resets counter with AdminCap → value becomes 0 ✓
```

Code Template

None

```
module counter::counter {
    use sui::event;

    // ================== Errors ==================

    const E_NOT_AUTHORIZED: u64 = 1;
    const E_INVALID_AMOUNT: u64 = 2;

    // ================== Structs ==================

    /// Admin capability - proves ownership
    public struct AdminCap has key, store {
        id: UID,
        counter_id: ID, // Which counter this cap controls
    }

    /// Shared counter object
    public struct Counter has key {
        id: UID,
        owner: address,
        value: u64,
    }

    // ================== Events ==================
```

```
    public struct CounterCreated has copy, drop {
        counter_id: ID,
        owner: address,
    }

    public struct CounterIncremented has copy, drop {
        counter_id: ID,
        old_value: u64,
        new_value: u64,
        incrementer: address,
    }

    public struct CounterReset has copy, drop {
        counter_id: ID,
        old_value: u64,
        new_value: u64,
    }

    // ==================== Functions ====================

    /// Create a new counter - returns AdminCap to creator
    public entry fun create_counter(ctx: &mut TxContext) {
        // TODO: Create counter object
        // TODO: Create AdminCap
        // TODO: Share counter
        // TODO: Transfer AdminCap to sender
        // TODO: Emit CounterCreated event
    }

    /// Increment counter by 1
    public entry fun increment(counter: &mut Counter, ctx:
&TxContext) {
        // TODO: Increment value by 1
        // TODO: Emit CounterIncremented event
    }

    /// Increment counter by specified amount
    public entry fun increment_by(
```

```
    counter: &mut Counter,
    amount: u64,
    ctx: &TxContext
) {
    // TODO: Validate amount > 0
    // TODO: Increment value by amount
    // TODO: Emit CounterIncremented event
}

/// Get current counter value (read-only)
public fun get_value(counter: &Counter): u64 {
    // TODO: Return current value
    counter.value
}

/// Reset counter to 0 (admin only)
public entry fun reset(
    _admin_cap: &AdminCap,
    counter: &mut Counter,
) {
    // TODO: Verify AdminCap belongs to this counter
    // TODO: Store old value
    // TODO: Reset value to 0
    // TODO: Emit CounterReset event
}

/// Reset counter to specific value (admin only)
public entry fun reset_to(
    _admin_cap: &AdminCap,
    counter: &mut Counter,
    new_value: u64,
) {
    // TODO: Verify AdminCap belongs to this counter
    // TODO: Store old value
    // TODO: Set value to new_value
    // TODO: Emit CounterReset event
}

// ================== Tests ==================
```

```
    #[test_only]
    public fun init_for_testing(ctx: &mut TxContext) {
        // Helper function for tests
    }

    #[test]
    fun test_create_and_increment() {
        use sui::test_scenario;

        let owner = @0xA;
        let mut scenario = test_scenario::begin(owner);

        // Create counter
        {
            create_counter(test_scenario::ctx(&mut scenario));
        };

        // Increment by user
        test_scenario::next_tx(&mut scenario, @0xB);
        {
            let mut counter =
test_scenario::take_shared<Counter>(&scenario);
            increment(&mut counter, test_scenario::ctx(&mut
scenario));
            assert!(get_value(&counter) == 1, 0);
            test_scenario::return_shared(counter);
        };

        test_scenario::end(scenario);
    }

    #[test]
    #[expected_failure(abort_code = E_INVALID_AMOUNT)]
    fun test_increment_zero_fails() {
        // TODO: Test that increment_by(0) fails
    }

    #[test]
```

```
    fun test_reset_by_admin() {
        // TODO: Test that admin can reset
    }

    #[test]
    #[expected_failure]
    fun test_reset_without_admin_cap_fails() {
        // TODO: Test that non-admin cannot reset
    }
}
```

Move.toml Template

```
None
[package]
name = "counter"
version = "0.0.1"
edition = "2024.beta"

[dependencies]
Sui = { git = "https://github.com/MystenLabs/sui.git", subdir
= "crates/sui-framework/packages/sui-framework", rev =
"framework/testnet" }

[addresses]
counter = "0x0"
```

Success Criteria

- [x] Create counter with initial value 0
- [x] Any user can increment counter
- [x] Counter value increases correctly
- [x] Admin can reset counter with AdminCap
- [x] Non-admin cannot reset counter (transaction fails)
- [x] Events are emitted for all actions
- [x] All tests pass

Technical Hints

- Use `share_object()` to make counter accessible to everyone

- `AdminCap` should be transferred to creator with `transfer::public_transfer()`
- Store counter's ID in AdminCap to verify ownership
- Use `assert!()` for validation
- Events should include relevant data for off-chain tracking

Challenge 2: Simple NFT Minting

Problem Description

Create a basic NFT collection where users can mint NFTs with customizable metadata. Each NFT is unique, transferable, and stores information like name, description, and image URL.

What You Need to Build

Core Functionality:

1. **NFT Structure**
    - Unique ID for each NFT
    - Name and description
    - Image URL
    - Creator address
    - Creation timestamp
2. **Minting Function**
    - Anyone can mint an NFT
    - Specify name, description, and image URL
    - NFT automatically transferred to minter
    - Emit minting event
3. **View Functions**
    - Get NFT name
    - Get NFT description
    - Get NFT image URL
    - Get creator address
4. **Transfer Function**
    - Owner can transfer NFT to another address
    - Emit transfer event
5. **Burn Function** (Optional)
    - Owner can destroy their NFT
    - Emit burn event

Expected Behavior

```
None
Day 1:
- Alice mints NFT: "Cool Cat #1", "A very cool cat",
"ipfs://..."
- Alice receives NFT object
- NFT ID: 0x123...
- Creator: Alice's address

Day 2:
- Bob mints NFT: "Cool Cat #2", "Another cool cat",
"ipfs://..."
- Bob receives NFT object

Day 3:
- Alice transfers NFT #1 to Charlie
- Charlie now owns NFT #1
- Creator still shows as Alice

Day 4:
- Charlie burns NFT #1
- NFT #1 is destroyed permanently
```

Code Template

```
None
module nft::simple_nft {
    use std::string::{Self, String};
    use sui::event;
    use sui::url::{Self, Url};

    // ==================== Errors ====================

    const E_NOT_OWNER: u64 = 1;
    const E_INVALID_NAME: u64 = 2;

    // ==================== Structs ====================

    /// The NFT object
    public struct SimpleNFT has key, store {
```

```
    id: UID,
    name: String,
    description: String,
    image_url: Url,
    creator: address,
    created_at: u64,
}

// ==================== Events ====================

public struct NFTMinted has copy, drop {
    nft_id: ID,
    name: String,
    creator: address,
    recipient: address,
}

public struct NFTTransferred has copy, drop {
    nft_id: ID,
    from: address,
    to: address,
}

public struct NFTBurned has copy, drop {
    nft_id: ID,
    burned_by: address,
}

// ==================== Functions ====================

/// Mint a new NFT
public entry fun mint_nft(
    name: vector<u8>,
    description: vector<u8>,
    image_url: vector<u8>,
    ctx: &mut TxContext
) {
    // TODO: Validate name is not empty
    // TODO: Create String from vectors
```

```
        // TODO: Create URL from vector
        // TODO: Create NFT object
        // TODO: Get sender address
        // TODO: Transfer NFT to sender
        // TODO: Emit NFTMinted event
    }

    /// Transfer NFT to another address
    public entry fun transfer_nft(
        nft: SimpleNFT,
        recipient: address,
        ctx: &TxContext
    ) {
        // TODO: Get NFT ID before transfer
        // TODO: Get sender address
        // TODO: Transfer NFT to recipient
        // TODO: Emit NFTTransferred event
    }

    /// Burn/destroy NFT
    public entry fun burn_nft(
        nft: SimpleNFT,
        ctx: &TxContext
    ) {
        // TODO: Get NFT ID
        // TODO: Get sender address
        // TODO: Unpack and destroy NFT
        // TODO: Emit NFTBurned event
    }

    // ==================== View Functions ====================

    /// Get NFT name
    public fun name(nft: &SimpleNFT): &String {
        &nft.name
    }

    /// Get NFT description
    public fun description(nft: &SimpleNFT): &String {
```

```
    &nft.description
}

/// Get NFT image URL
public fun image_url(nft: &SimpleNFT): &Url {
    &nft.image_url
}

/// Get NFT creator
public fun creator(nft: &SimpleNFT): address {
    nft.creator
}

/// Get creation timestamp
public fun created_at(nft: &SimpleNFT): u64 {
    nft.created_at
}

// ==================== Tests ====================

#[test]
fun test_mint_nft() {
    use sui::test_scenario;

    let owner = @0xA;
    let mut scenario = test_scenario::begin(owner);

    // Mint NFT
    {
        mint_nft(
            b"Cool Cat #1",
            b"A very cool cat",
            b"ipfs://QmX...",
            test_scenario::ctx(&mut scenario)
        );
    };

    // Check NFT was created
    test_scenario::next_tx(&mut scenario, owner);
```

```
        {
            let nft =
test_scenario::take_from_sender<SimpleNFT>(&scenario);
            assert!(name(&nft) == &string::utf8(b"Cool Cat
#1"), 0);
            assert!(creator(&nft) == owner, 1);
            test_scenario::return_to_sender(&scenario, nft);
        };

        test_scenario::end(scenario);
    }

    #[test]
    fun test_transfer_nft() {
        // TODO: Test minting and transferring NFT
    }

    #[test]
    fun test_burn_nft() {
        // TODO: Test burning NFT
    }

    #[test]
    #[expected_failure(abort_code = E_INVALID_NAME)]
    fun test_mint_empty_name_fails() {
        // TODO: Test that empty name fails
    }
}
```

Move.toml Template

```
None
[package]
name = "simple_nft"
version = "0.0.1"
edition = "2024.beta"

[dependencies]
```

```
Sui = { git = "https://github.com/MystenLabs/sui.git", subdir
= "crates/sui-framework/packages/sui-framework", rev =
"framework/testnet" }

[addresses]
nft = "0x0"
```

Success Criteria

- [x] Mint NFT with custom metadata
- [x] NFT is transferred to minter
- [x] View functions return correct data
- [x] Transfer NFT to another address
- [x] Burn NFT successfully
- [x] Events emitted for mint, transfer, burn
- [x] All tests pass

Technical Hints

- Use `string::utf8()` to convert `vector<u8>` to `String`
- Use `url::new_unsafe_from_bytes()` to create URLs
- NFTs should have `key` ability (owned objects)
- Add `store` ability to make NFTs transferable
- Use `transfer::public_transfer()` for transfers
- Use `object::delete()` to destroy NFT's UID when burning

Challenge 3: Simple Staking Contract

Problem Description

Build a staking system where users can stake tokens for a fixed duration and earn rewards. Users lock their tokens for 30 days and receive 10% annual returns when they unstake.

What You Need to Build

Core Functionality:

1. **Staking Pool**
    - Shared pool to track all stakes
    - Store total staked amount

    - Track individual user stakes
2. **Stake Function**
    - User deposits tokens into pool
    - Record stake amount and timestamp
    - Tokens locked for 30 days minimum
    - Emit staking event
3. **Unstake Function**
    - User withdraws tokens after lock period
    - Calculate rewards based on time staked
    - Transfer original stake + rewards
    - Remove stake record
    - Emit unstake event
4. **View Functions**
    - Check if user has active stake
    - Get stake amount
    - Get stake timestamp
    - Calculate pending rewards
    - Check if stake is unlocked
5. **Reward Calculation**
    - Formula: `reward = stake_amount * 10% * (days_staked / 365)`
    - Minimum lock period: 30 days
    - Maximum calculation: 365 days (even if staked longer)

Expected Behavior

None

```
Day 1:
- Alice stakes 1,000 tokens
- Stake locked until Day 31
- Lock timestamp recorded

Day 15:
- Alice tries to unstake → FAILS (only 15 days passed)
- Pending rewards ≈ 4.1 tokens (1000 * 10% * 15/365)

Day 31:
- Alice can now unstake
- Days staked: 30
- Reward: 1000 * 10% * 30/365 ≈ 8.2 tokens
- Alice receives: 1,000 + 8.2 = 1,008.2 tokens
```

```
Day 1 (Bob):
- Bob stakes 5,000 tokens

Day 366 (Bob):
- Bob unstakes after 365+ days
- Reward capped at 365 days: 5000 * 10% = 500 tokens
- Bob receives: 5,000 + 500 = 5,500 tokens
```

Code Template

None

```
module staking::simple_staking {
    use sui::coin::{Self, Coin};
    use sui::balance::{Self, Balance};
    use sui::sui::SUI;
    use sui::clock::{Self, Clock};
    use sui::table::{Self, Table};
    use sui::event;

    // ================== Constants ==================

    const LOCK_PERIOD_MS: u64 = 30 * 24 * 60 * 60 * 1000; //
30 days in milliseconds
    const ANNUAL_REWARD_RATE: u64 = 10; // 10% APY
    const DAYS_IN_YEAR: u64 = 365;
    const MS_IN_DAY: u64 = 24 * 60 * 60 * 1000;

    // ================== Errors ==================

    const E_STAKE_LOCKED: u64 = 1;
    const E_NO_STAKE: u64 = 2;
    const E_INSUFFICIENT_BALANCE: u64 = 3;
    const E_ZERO_AMOUNT: u64 = 4;
    const E_ALREADY_STAKED: u64 = 5;

    // ================== Structs ==================
```

```
/// Individual stake information
public struct StakeInfo has store {
    amount: u64,
    stake_timestamp: u64,
}

/// Shared staking pool
public struct StakingPool has key {
    id: UID,
    /// Pool balance holding staked tokens
    balance: Balance<SUI>,
    /// Tracks each user's stake
    stakes: Table<address, StakeInfo>,
    /// Total amount staked
    total_staked: u64,
}

// ================== Events ==================

public struct PoolCreated has copy, drop {
    pool_id: ID,
}

public struct Staked has copy, drop {
    staker: address,
    amount: u64,
    timestamp: u64,
}

public struct Unstaked has copy, drop {
    staker: address,
    amount: u64,
    reward: u64,
    timestamp: u64,
}

// ================== Functions ==================

/// Initialize the staking pool
```

```
    fun init(ctx: &mut TxContext) {
        // TODO: Create StakingPool with empty balance and
table
        // TODO: Share the pool object
        // TODO: Emit PoolCreated event
    }

    /// Stake tokens
    public entry fun stake(
        pool: &mut StakingPool,
        stake_coin: Coin<SUI>,
        clock: &Clock,
        ctx: &mut TxContext
    ) {
        // TODO: Validate amount > 0
        // TODO: Check user doesn't already have a stake
        // TODO: Get stake amount
        // TODO: Add coin balance to pool
        // TODO: Create StakeInfo record
        // TODO: Add to stakes table
        // TODO: Update total_staked
        // TODO: Emit Staked event
    }

    /// Unstake tokens and claim rewards
    public entry fun unstake(
        pool: &mut StakingPool,
        clock: &Clock,
        ctx: &mut TxContext
    ) {
        // TODO: Get sender address
        // TODO: Check stake exists
        // TODO: Get stake info
        // TODO: Check lock period has passed
        // TODO: Calculate rewards
        // TODO: Calculate total payout (stake + rewards)
        // TODO: Check pool has sufficient balance
        // TODO: Take coins from pool balance
        // TODO: Remove stake from table
```

```
        // TODO: Update total_staked
        // TODO: Transfer coins to user
        // TODO: Emit Unstaked event
    }

    // ================== Helper Functions
==================

    /// Calculate reward for a stake
    public fun calculate_reward(
        amount: u64,
        stake_timestamp: u64,
        current_timestamp: u64
    ): u64 {
        // TODO: Calculate time elapsed in milliseconds
        // TODO: Convert to days
        // TODO: Cap at 365 days
        // TODO: Calculate reward: amount * rate * (days /
365)
        // TODO: Return reward
        0 // Placeholder
    }

    /// Check if stake can be unlocked
    public fun is_unlocked(
        stake_timestamp: u64,
        current_timestamp: u64
    ): bool {
        // TODO: Check if lock period has passed
        current_timestamp >= stake_timestamp + LOCK_PERIOD_MS
    }

    // ================== View Functions ==================

    /// Check if user has active stake
    public fun has_stake(pool: &StakingPool, user: address):
bool {
        table::contains(&pool.stakes, user)
    }
```

```
    /// Get user's stake amount
    public fun get_stake_amount(pool: &StakingPool, user:
address): u64 {
        if (!table::contains(&pool.stakes, user)) {
            return 0
        };
        let stake_info = table::borrow(&pool.stakes, user);
        stake_info.amount
    }

    /// Get user's stake timestamp
    public fun get_stake_timestamp(pool: &StakingPool, user:
address): u64 {
        // TODO: Return stake timestamp or 0 if no stake
        0 // Placeholder
    }

    /// Get pending rewards
    public fun get_pending_rewards(
        pool: &StakingPool,
        user: address,
        clock: &Clock
    ): u64 {
        // TODO: Check if user has stake
        // TODO: Get stake info
        // TODO: Calculate and return rewards
        0 // Placeholder
    }

    /// Get total staked in pool
    public fun get_total_staked(pool: &StakingPool): u64 {
        pool.total_staked
    }

    // ==================== Admin Functions ====================

    /// Add funds to pool (for rewards)
    public entry fun add_rewards(
```

```
        pool: &mut StakingPool,
        reward_coin: Coin<SUI>,
        _ctx: &mut TxContext
    ) {
        // TODO: Add coin to pool balance
        // Note: Don't update total_staked (these are rewards,
not stakes)
    }

    // ================== Tests ==================

    #[test_only]
    public fun init_for_testing(ctx: &mut TxContext) {
        init(ctx);
    }

    #[test]
    fun test_stake_and_unstake() {
        use sui::test_scenario;
        use sui::test_utils;

        let admin = @0xAD;
        let alice = @0xA;

        let mut scenario = test_scenario::begin(admin);

        // Initialize pool
        {
            init_for_testing(test_scenario::ctx(&mut
scenario));
        };

        // Alice stakes
        test_scenario::next_tx(&mut scenario, alice);
        {
            let mut pool =
test_scenario::take_shared<StakingPool>(&scenario);
            let clock =
clock::create_for_testing(test_scenario::ctx(&mut scenario));
```

```
            let stake_coin = coin::mint_for_testing<SUI>(1000,
test_scenario::ctx(&mut scenario));
            stake(&mut pool, stake_coin, &clock,
test_scenario::ctx(&mut scenario));

            assert!(get_stake_amount(&pool, alice) == 1000,
0);

            clock::destroy_for_testing(clock);
            test_scenario::return_shared(pool);
        };

        // Try to unstake immediately (should fail)
        test_scenario::next_tx(&mut scenario, alice);
        {
            let mut pool =
test_scenario::take_shared<StakingPool>(&scenario);
            let clock =
clock::create_for_testing(test_scenario::ctx(&mut scenario));

            // This should fail due to lock period
            // unstake(&mut pool, &clock,
test_scenario::ctx(&mut scenario));

            clock::destroy_for_testing(clock);
            test_scenario::return_shared(pool);
        };

        test_scenario::end(scenario);
    }

    #[test]
    fun test_reward_calculation() {
        // TODO: Test reward calculation formula
        // 1000 tokens * 10% * 30 days / 365 days ≈ 8.2 tokens
        let amount = 1000;
        let start = 0;
        let end = 30 * MS_IN_DAY;
```

```
        let reward = calculate_reward(amount, start, end);
        // Allow some rounding error
        assert!(reward >= 8 && reward <= 9, 0);
    }

    #[test]
    #[expected_failure(abort_code = E_STAKE_LOCKED)]
    fun test_cannot_unstake_before_lock_period() {
        // TODO: Test that unstaking before 30 days fails
    }

    #[test]
    #[expected_failure(abort_code = E_ALREADY_STAKED)]
    fun test_cannot_stake_twice() {
        // TODO: Test that user cannot stake twice
simultaneously
    }
}
```

Move.toml Template

```
None
[package]
name = "simple_staking"
version = "0.0.1"
edition = "2024.beta"

[dependencies]
Sui = { git = "https://github.com/MystenLabs/sui.git", subdir
= "crates/sui-framework/packages/sui-framework", rev =
"framework/testnet" }

[addresses]
staking = "0x0"
```

Success Criteria

- [x] Create staking pool

- [x] User can stake tokens
- [x] Stake is locked for 30 days
- [x] Cannot unstake before lock period
- [x] Rewards calculated correctly (10% APY)
- [x] User receives stake + rewards on unstake
- [x] Pool tracks total staked amount
- [x] Events emitted for stake/unstake
- [x] All tests pass

Technical Hints

- Use `Balance<T>` to hold tokens in the pool
- Use `Table` to track individual stakes (efficient lookup)
- Use `Clock` for all time-based checks
- Calculate rewards using integer math (avoid decimals)
- Consider scaling values to avoid precision loss: multiply before dividing
- Test edge cases: stake for exactly 30 days, 365+ days, 0 days
- Use `coin::take()` to extract coins from Balance
- Remember to update `total_staked` on both stake and unstake

Common Testing Patterns

Testing with Clock

```
None
#[test]
fun test_with_time() {
    use sui::test_scenario;
    use sui::clock;

    let mut scenario = test_scenario::begin(@0x1);

    {
        let mut clock =
clock::create_for_testing(test_scenario::ctx(&mut scenario));

        // Start at timestamp 0
        assert!(clock::timestamp_ms(&clock) == 0, 0);

        // Fast forward 30 days
```

```
        clock::increment_for_testing(&mut clock, 30 * 24 * 60
* 60 * 1000);

        assert!(clock::timestamp_ms(&clock) == 30 * 24 * 60 *
60 * 1000, 1);

        clock::destroy_for_testing(clock);
    };

    test_scenario::end(scenario);
}
```

Testing Shared Objects

None

```
#[test]
fun test_shared_object() {
    use sui::test_scenario;

    let admin = @0xAD;
    let user = @0xA;

    let mut scenario = test_scenario::begin(admin);

    // Admin creates shared object
    {
        create_shared_object(test_scenario::ctx(&mut
scenario));
    };

    // User interacts with shared object
    test_scenario::next_tx(&mut scenario, user);
    {
        let mut obj =
test_scenario::take_shared<SharedObject>(&scenario);

        // Do something with obj
        interact_with(&mut obj);
```

```
        test_scenario::return_shared(obj);
    };

    test_scenario::end(scenario);
}
```

Testing Owned Objects

```
None
#[test]
fun test_owned_object() {
    use sui::test_scenario;

    let user = @0xA;
    let mut scenario = test_scenario::begin(user);

    // Create object for user
    {
        create_owned_object(test_scenario::ctx(&mut
scenario));
    };

    // User accesses their object
    test_scenario::next_tx(&mut scenario, user);
    {
        let obj =
test_scenario::take_from_sender<OwnedObject>(&scenario);

        // Check properties
        assert!(get_value(&obj) == 42, 0);

        test_scenario::return_to_sender(&scenario, obj);
    };

    test_scenario::end(scenario);
}
```

README Template

None

```
# [Challenge Name]

## Overview
[Brief description of what your contract does]

## Features
- Feature 1
- Feature 2
- Feature 3

## Project Structure
```
├── sources/
│   └── [module_name].move
├── tests/
│   └── [module_name]_tests.move
└── Move.toml
```

## Building

```bash
sui move build
```

Testing

```bash
sui move test
```

For verbose output:
```bash
sui move test --verbose
```

Deployment
```

```bash
sui client publish --gas-budget 100000000
```

Usage Examples

Example 1: [Action Name]
```bash
sui client call \
  --package [PACKAGE_ID] \
  --module [MODULE_NAME] \
  --function [FUNCTION_NAME] \
  --args [ARGS] \
  --gas-budget 10000000
```

Example 2: [Another Action]
```bash
# Command here
```

Contract Functions

Public Entry Functions

`function_name`
- **Description**: What it does
- **Parameters**:
  - `param1`: Description
  - `param2`: Description
- **Access**: Who can call it
- **Example**: Usage example

View Functions

`view_function_name`
- **Description**: What it returns
- **Parameters**: What it needs

```
- **Returns**: What you get back

## Deployed Contract (Testnet)

- **Package ID**: `0x...`
- **Object ID**: `0x...`
- **Network**: Sui Testnet

## Design Decisions

### Why [Choice 1]?
Explanation of why you made this design decision.

### Why [Choice 2]?
Another design decision explanation.

## Testing Coverage

- [x] Happy path scenarios
- [x] Error cases
- [x] Edge cases
- [x] Access control
- [x] Time-based logic (if applicable)

## Known Limitations

- Limitation 1
- Limitation 2

## Future Improvements

- [ ] Improvement 1
- [ ] Improvement 2

## Resources

- [Sui Documentation](https://docs.sui.io)
- [Move Language Book](https://move-language.github.io/move/)
```

```
- [Sui
Examples](https://github.com/MystenLabs/sui/tree/main/examples
)
```

Evaluation Criteria

- **Completeness** : All required functions implemented
- **Correctness** : Code compiles and works as expected
- **Code Quality** : Clean, readable code with comments with atleast 4 commits.
- **Submissions:** Code should be deployed to testnet and should share the package ID with valid test results
    - **Need to share the published packaged id in discord (https://discord.com/invite/sQEkBSZbd7)**

Support & Resources

- **Sui Discord**: https://discord.gg/sui
- **Sui Forum**: https://forums.sui.io
- **Documentation**: https://docs.sui.io
- **Move Examples**: https://examples.sui.io

Good luck with your challenges! 🚀