**Lesson 3: Understanding Fungible/Non-fungible tokens**

**Lessson Summary**

* This lesson introduces the concept of cryptographic tokens
* We explain Fungible and Non-Fungible tokens (NFTs) and their main functions
* We also explain the difference between fungible and non-fungible tokens (NFTs)
* The importance of understanding these cryptographic concepts is emphasized, especially in relation to creating programs

**Lesson Objective**

By the end of the lesson, you will:

* Define cryptographic tokens and distinguish them from other types of tokens
* Differentiate between fungible and non-fungible tokens and explain their respective use cases
* Apply their knowledge of cryptographic tokens to create programs using both fungible and non-fungible tokens
* Understand how to implement fungible and non-fungible tokens in the Gear Protocol

**Let’s get started!**

**Fungible token & their Properties**

Fungible tokens offer the same value and exchangeability as fiat currency. Much like exchanging one paper dollar for another, these digital smart contracts allow users to trade tokenized assets of equal worth between accounts. However, at a fundamental technological level, fungible tokens are simply smart contracts that store a mapping between account addresses and the number of tokens.

*аddress <math xmlns="http://www.w3.org/1998/Math/MathML"><mo>&#x2192;</mo></math> amount*

The main function of such smart contracts are:

* *Transfer(from, to, amount):* This function allows you to transfer the number of tokens (amount) from one address (from) to another (to). It checks if the “from” account owns tokens, subtracts the necessary amount from its balance, and adds the specified token number to the “to” account*.*
* *Approve(spender, amount)* is a function that allows you to give the specified *spender* account the right to dispose of the tokens of the account that called this function (in our case, it'll be *msg::source()*). In the other words, the *spender* account can call the *transfer()* function, so it can transfer tokens from the *msg::source()* account to the specified address. This functionality is useful when the transfer of tokens occurs in any of the contracts.

Let's take an *escrow* smart contract as an example.

In this example, the goods are paid using tokens and not a *msg::value()*. The buyer sends a *deposit()* message, and the *escrow* smart contract accesses the token contract and sends a token transfer message. In this particular message, the *from* address is the buyer address.

If the *escrow* contract does not have the right to dispose of the buyer's tokens, then the token contract will panic and prevent the token transfer.

* *Mint(to, amount):* This function increases the number of tokens in the contract. Usually, this function can be called by certain accounts that are allowed to create new tokens*.*
* *Burn(from, amount)* is a function that reduces the token number in the contract. Just like with the *mint()* function, not all accounts are allowed to burn tokens.

**Non-Fungible Tokens**

Non-fungible tokens, or NFTs, offer a unique way to prove digital asset ownership. While traditional fungible tokens are interchangeable and store a value, NFTs carry cryptographic certificates that demonstrate the owner's authority over an asset, like digital art or gaming assets.

*аddress <math xmlns="http://www.w3.org/1998/Math/MathML"><mo>&#x2192;</mo></math>token\_id*

The main functions of the contract of such tokens are similar to fungible tokens:

* *Transfer(to, token\_id)* is a function that allows you to transfer a token with the *token\_id* number to the to account. Unlike the fungible token contract, this contract does not require a *from* the account, since each token has its own owner.
* *Approve(approved\_account, token\_id)* is a function that allows you to give the right to dispose of the token to the specified *approved\_account*. This functionality can be useful on marketplaces for auctions. When the owner wants to sell his token, they can put it on a marketplace/auction, so the contract sends this token to the new owner.
* *Mint(to, token\_id, metadata)* isa function that creates a new token. *Metadata* can include any information about the token: it can be a link to a specific resource, a description of the token, etc.
* *Burn(from, token\_id):* This function removes the token with the mentioned *token\_id* from the contract*.*

**Asynchronous communication between programs**

One of the key and distinguished features of the Gear Protocol is the Actor model for message-passing communications. Gear Protocol leverages the Actor model for message-passing communication, allowing parallel computation and asynchronous messaging to ensure faster processing times. The development structure provides developers with immense flexibility when building complex dApps.

If a program sends an asynchronous message to another program, it needs to wait for the reply from that program before it can proceed to the next operation.

To send a message to a Gear program, we use the send\_for\_reply(program, payload, value) function. In this function:

* program - the address of the program to send the message for;
* payload - the message to the program;
* value - the funds attached to the message.

| pub fn **send\_for\_reply\_as**<E: Encode, D: Decode>(  program: ActorId,  payload: E,  value: u128 ) -> Result<CodecMessageFuture<D>> |
| --- |

**Distributed transactions**

Interactions between programs in the Gear Protocol create distributed transactions that involve operations across actors with their respective states. In our case, operations are performed across actors with their states. The distributed transactions must possess the following features:

* Atomicity: All data changes are treated as if they were a single operation. That is, either all of the modifications are made or none.
* Consistency: This property implies that when a transaction begins and ends, the state of data is consistent.

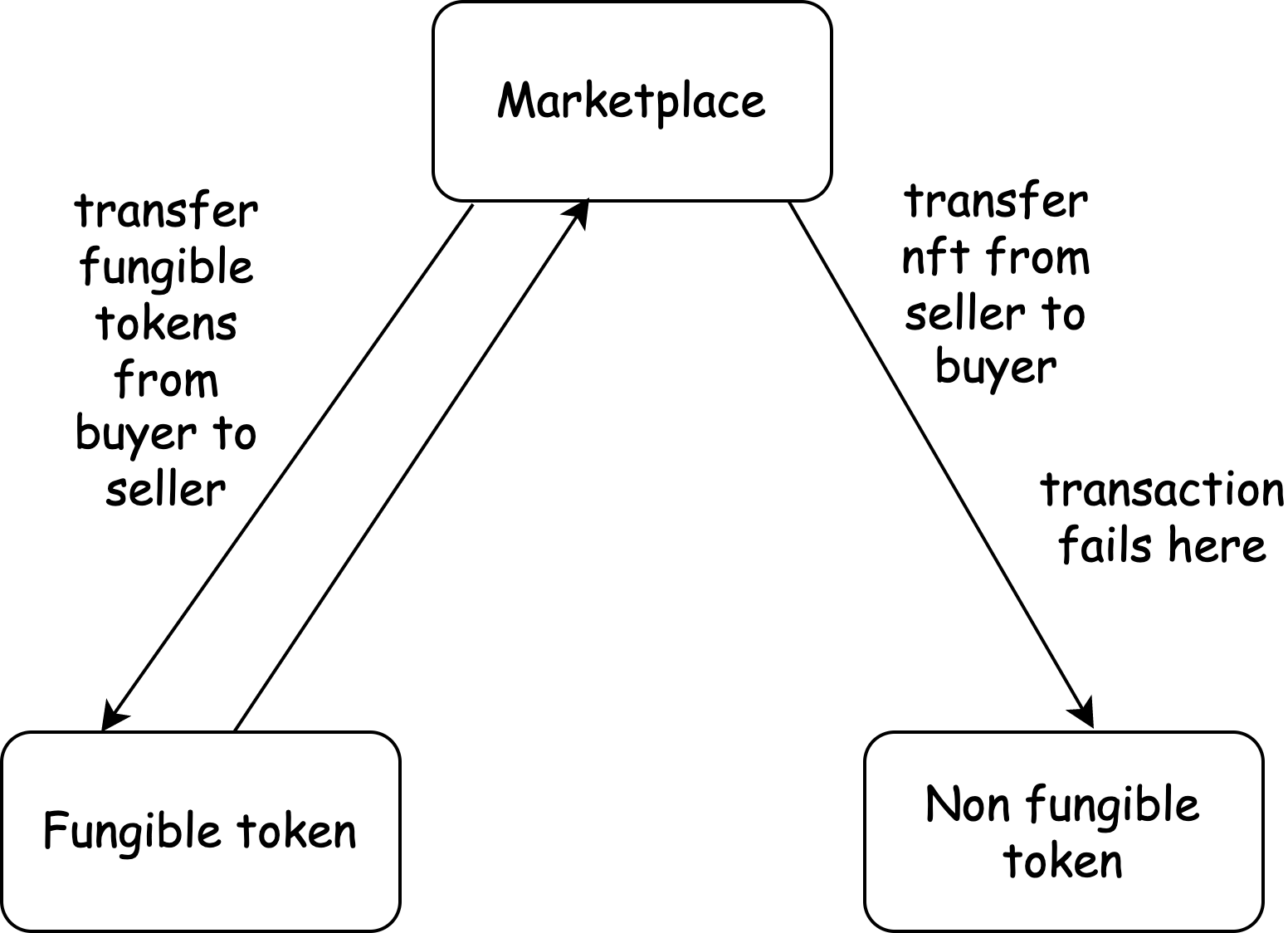
For instance, in Ethereum transactions, global state changes only occur when all executions finish successfully. If an error occurs during execution, all changes to the state are "rolled back," as if the transaction had never been running.

Let’s look at the following code:

| static mut COUNTER: u32 = 0;  async unsafe fn **non\_atomic**() {  COUNTER = 10;   send\_for\_reply(msg::source(), "PING", 0)  .expect("Error during sending message")  .await  .expect("Error during message execution");   COUNTER = 20; } |
| --- |

In the example code provided, the global variable COUNTER is set to 10 before the *send\_for\_reply* function is called. If the transaction fails before *.await*, the state is rolled back, and COUNTER returns to 0. If the transaction fails after *.await*, COUNTER retains its value of 10.

Let’s consider an example of a simple marketplace where tokens are transferred to the seller, and then transfers NFT to the buyer.



The picture shows the following situation:

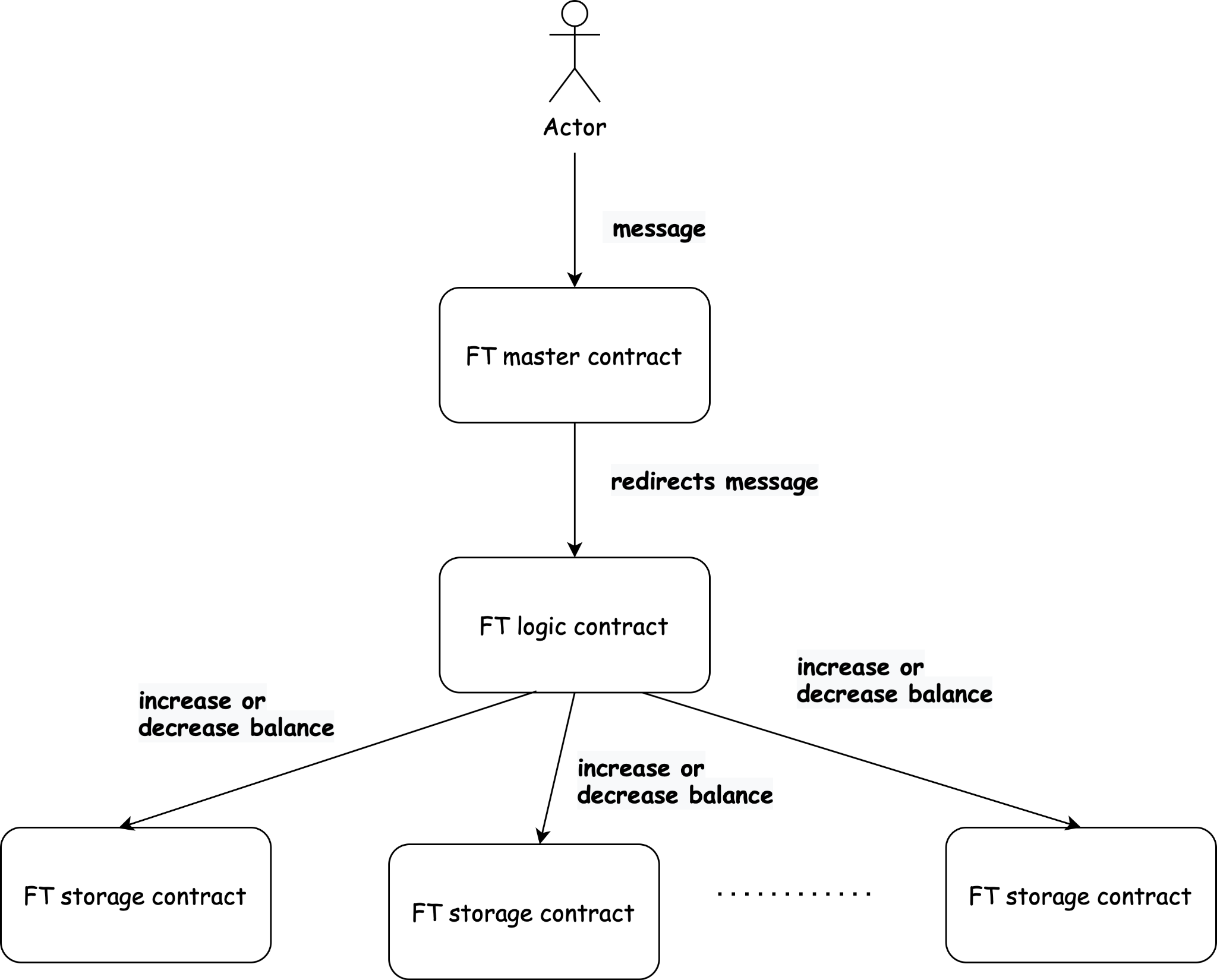
1. The marketplace successfully transfers tokens to the seller;
2. During the NFT transfer to the buyer, the transaction fails.

The failed transaction during the transfer of NFTs from the seller to the buyer after the successful transfer of tokens would result in an inconsistent state, with the seller receiving payment but the buyer not receiving the NFT. Thus, we must consider potential failures leading to state inconsistency when developing applications and different standards.

**Implementation of Fungible Tokens on Gear**

We propose to split the fungible token into three contracts:

1. The **master** fungible token that serves as a proxy program that redirects the message to the logic contract.
2. The **Token Logic Contract** - responsible for realizing the main standard token functions. We place the logic in a separate contract to add more functions without losing the address of the fungible token and the contract state.;
3. **Storage Contracts**: These contracts store the balances of the users.



The token standard has a feature of Preventing Duplicate Transaction (Maintaining idempotency): There are two possible risks when sending a transaction:

* Sending duplicate transactions
* Not knowing the transaction status due to a network failure.

The sender can be assured that the transaction will only be executed once (idempotency).

**Storage contract architecture**

The storage contracts state has the following fields:

* The address of the logic contract. The storage contract must execute messages received only from that address;

| ft\_logic\_id: ActorId |
| --- |

* The executed transactions. In each message, the storage contract receives the hash of the transaction that is being executed and stores its execution results in the field Executed. If Executed is true, then the message executed successfully, otherwise Executed equals false.

| transaction\_status: HashMap<H256, (Executed, Locked)> |
| --- |

* Balances of accounts

| balances: HashMap<ActorId, u128> |
| --- |

* Approvals of accounts

| approvals: HashMap<ActorId, HashMap<ActorId, u128>> |
| --- |

The messages that the storage accepts:

* Increase balance: the storage raises the balance of the indicated account;
* Decrease balance: The storage reduces the balance of the indicated account;
* Approve:The storage allow the account to give another account permission to transfer his tokens;
* Transfer: Transfer tokens from one account to another. The message is called from the logic contract when the token transfer occurs in one storage.
* Clear: Remove the hash of the executed transaction.

That storage contract doesn't make any asynchronous calls, so its execution is atomic.

**The logic contract architecture**

The state of the logic contract consists of the following fields:

* **The master token contract address.** The logic contract must execute messages only from that address:

| ftoken\_id: ActorId |
| --- |

* **The transactions**. As in the storage contract, the logic contract receives the hash of the transaction that is being executed and stores the result of its execution. But unlike the storage contract, where message executions are atomic, the logic contract has to keep track of the message being executed and its stage.

| transactions: HashMap<H256, Transaction> |
| --- |

The Transaction is the following struct:

| pub struct Transaction {  msg\_source: ActorId,  operation: Operation,  status: TransactionStatus, } |
| --- |

Where msg\_source is an account that sends a message to the main contract. Operation is the action that the logic contract should process and status is the transaction status. it's the following enum.

| pub enum **TransactionStatus** {  InProgress,  Success,  Failure, } |
| --- |

* InProgress - the transaction execution started;
* Success or Failure - the transaction was completed (successfully or not). In this case, the logic contract only sends a response that the transaction with this hash has already been completed.
* **The code hash of the storage contract**. The logic contract is able to create a new storage contract when it's necessary. The storage creation is implemented as follows:
  + The logic contract takes the first letter of the account address. If the storage contract for this letter is created, then it stores the balance of this account in this contract. If not, it creates a new storage contract

| storage\_code\_hash: H256 |
| --- |

* **The mapping from letters to the storage addresses.**

| id\_to\_storage: HashMap<String, ActorId> |
| --- |

The logic contract receives from the master contract the following message:

| Message {  transaction\_hash: H256,  account: ActorId,  payload: Vec<u8>,  }, |
| --- |

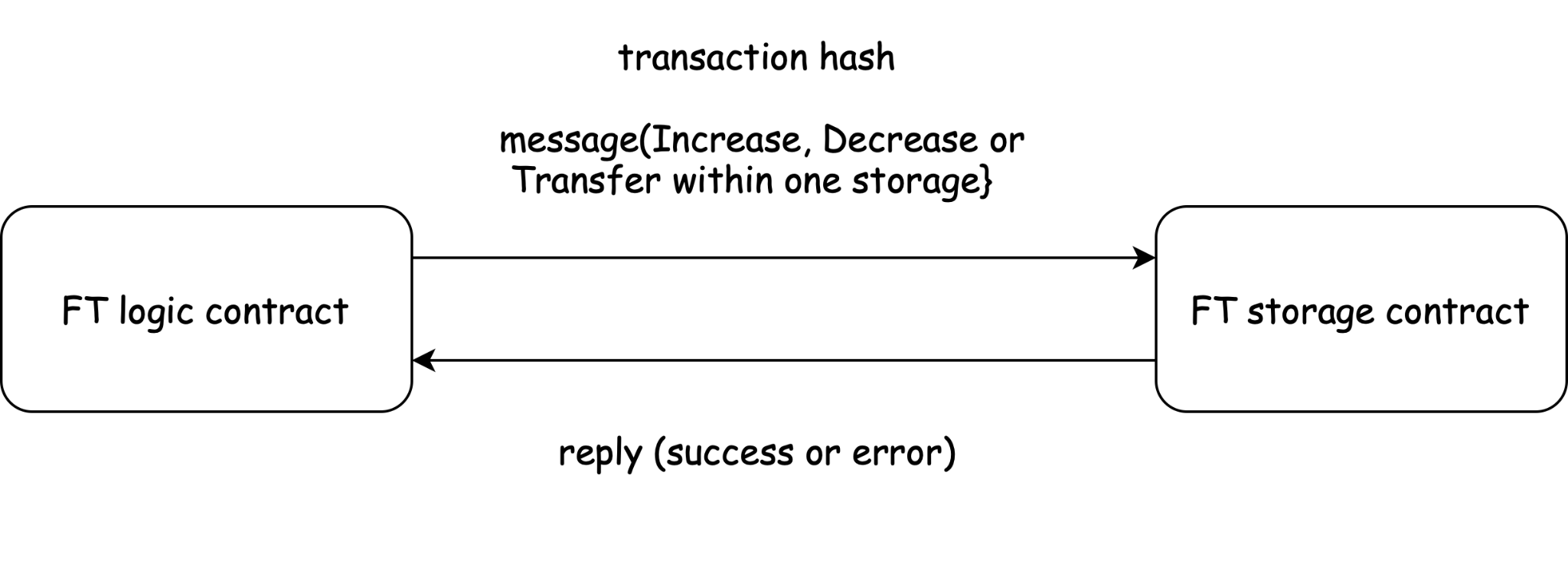
The account is an actor who sends the message to the master contract.

The payload is the encoded operation the logic contract has to process:

| pub enum Operation {  Mint {  recipient: ActorId,  amount: u128,  },  Burn {  sender: ActorId,  amount: u128,  },  Transfer {  sender: ActorId,  recipient: ActorId,  amount: u128,  },  Approve {  approved\_account: ActorId,  amount: u128,  }, } |
| --- |

When upgrading the logic contract, there may be changes to the enum Operation, which means the payload structure may also change. As a result, the master contract does not know the specific type of payload structure and instead sends it as a byte array (Vec<u8>).

The logic contract sends only one message to the storage contract during the message Mint, Burn or Transfer between accounts in the same storage. Upon receiving the message, the logic contract decodes the payload from a byte array into the expected enum Operation. This allows the logic contract to process the message based on the specific operation type (Mint, Burn, orTransfer)



When the transfer occurs between 2 different storages, the contract acts as follows:

1. The logic contract sends the DecreaseBalance message to the storage contract.
2. If the message executes successfully, the logic contract sends the message IncreaseBalance to another storage contract. Otherwise, the logic contract saves the status failure and replies to the main contract.
3. If the message IncreaseBalance executes successfully, the logic contract saves the status and replies to the main contract. If the gas ran out during the IncreaseBalance execution in the storage contract, the logic contract saves the status DecreaseSuccess. This status is untrackable in the handle\_signal function.

If a transaction has been executed unsuccessfully, it could be due to an issue with the contract memory. The logic contract must trace storage contracts and re-run any failed transactions to prevent failure. If the errors persist, then the balance should be returned.

**The master contract architecture**

The state of the master contract includes the following fields:

* The address of the contract admin. He has the right to upgrade the logic contract.

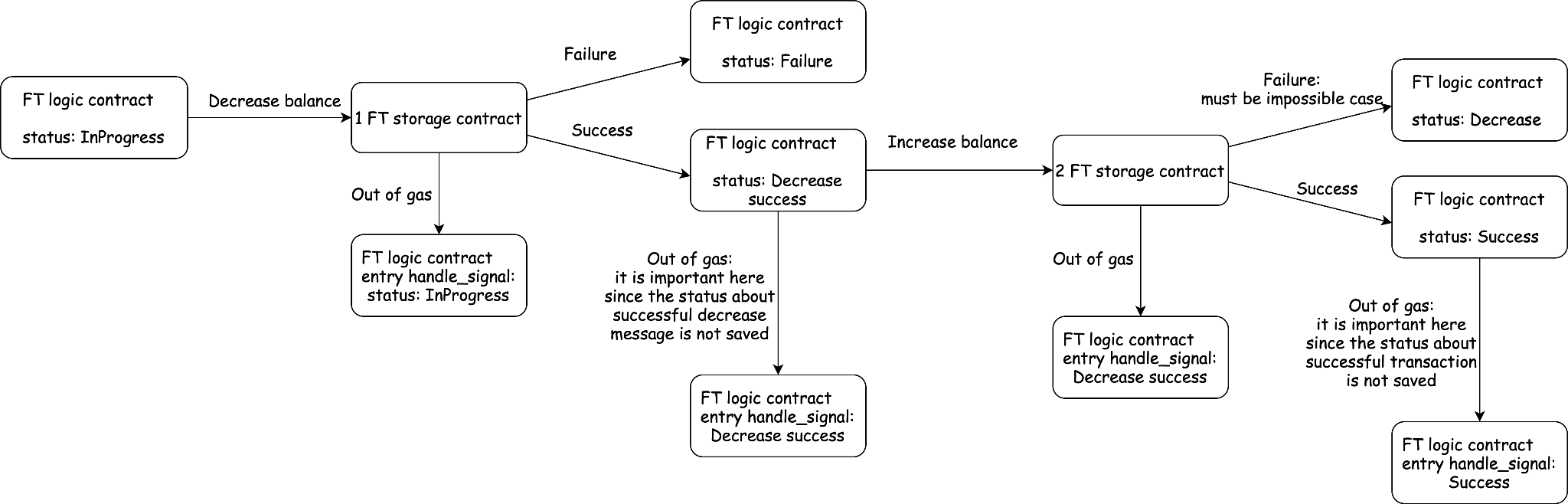
| admin: ActorId, |
| --- |

* The address of the logic contract

| ft\_logic\_id: ActorId, |
| --- |

* The transaction history.

| transactions: HashMap<H256, TransactionStatus> |
| --- |

Where the TransactionStatus:

| pub enum **TransactionStatus** {  InProgress,  Success,  Failure, } |
| --- |

The contract receives a message from an account with a specific nonce, which is used to compute the transaction hash, along with the account address. It is the user's responsibility to keep track of their nonce and increase it with each subsequent transaction. However, it is possible to design the contract in a way that automatically tracks the user's nonce, making the nonce field optional.

The main contract just redirects that message to the logic contract indicating the account that sends a message to it.

**Assignment**

In this assignment, you will add the functionality to your Tamagotchi smart contract to allow for changing ownership and approving other accounts to change ownership. This will involve implementing the following functions:

* Transfer(new\_owner) - that action must change the field owner to the indicated account;
* Approve(allowed\_account) - that action must fill the field approved\_account the indicated account;
* RevokeApproval - that action removes the current approved\_account.

Upload your contract to the blockchain, run the frontend application and select the third lesson.

To ensure that your contract is compatible with the frontend application, please make sure that the metadata is set to the following:

| pub struct ProgramMetadata;  impl Metadata for ProgramMetadata {  type Init = InOut<String, ()>;  type Reply = InOut<(), ()>;  type Others = InOut<(), ()>;  type Signal = ();  type Handle = InOut<TmgAction, TmgEvent>;  type State = Tamagotchi; }  *#[derive(Encode, Decode, TypeInfo)]* pub enum TmgAction {  Name,  Age,  Feed,  Play,  Sleep,  Transfer(ActorId),  Approve(ActorId),  RevokeApproval, }  *#[derive(Encode, Decode, TypeInfo)]* pub enum TmgEvent {  Name(String),  Age(u64),  Fed,  Entertained,  Slept,  Transfer(ActorId),  Approve(ActorId),  RevokeApproval, }  *#[derive(Default, Encode, Decode, TypeInfo)]* pub struct Tamagotchi {  pub name: String,  pub date\_of\_birth: u64,  pub owner: ActorId,  pub fed: u64,  pub fed\_block: u64,  pub entertained: u64,  pub entertained\_block: u64,  pub rested: u64,  pub rested\_block: u64,  pub allowed\_account: Option<ActorId>, } |
| --- |