**Lesson 4: The Tamagotchi Shop Description**

**Lesson Summary:**

* The lesson covers how to create contracts for selling Tamagotchi using fungible tokens.
* The Tamagotchi can have various attributes such as accessories, clothing, and weapons, which will be necessary for our upcoming Tamagotchi battle game.
* To buy attributes, the Tamagotchi must have enough tokens in its balance, and it must approve the store's contract for transferring its tokens.
* We’ll explore transaction handling and how to deal with transactions with errors during their execution.

**Lesson Objective:**

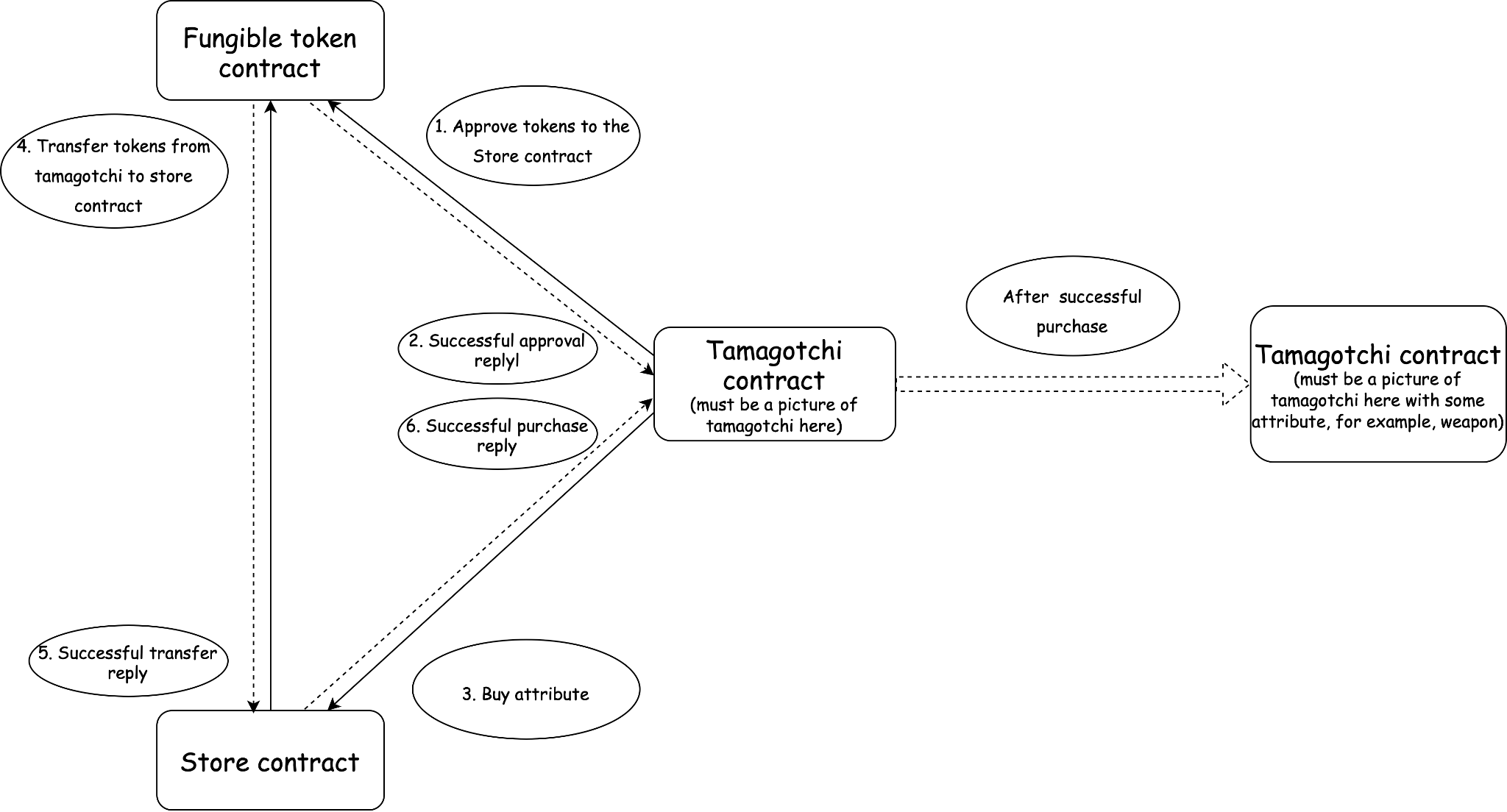
By the end of this lesson, users will:

* Create a contract for selling Tamagotchi using fungible tokens
* Connect the concept of fungible tokens (from Lesson 3) and their use in purchasing Tamagotchi attributes
* Understand how to check Tamagotchi token balance and approve contracts for transferring tokens.
* Understand how to handle incomplete transactions

**Let’s get started!**

The buying process involves three steps:

1. The Tamagotchi sends a message to the fungible token contract to approve the store contract to transfer its tokens;
2. The Tamagotchi sends a message to the store contract, indicating the attribute it wants to buy;
3. The store contract sends a message to the fungible token contract to transfer the tokens to itself. If the tokens are successfully transferred, the store adds the attribute to the the Tamagotchi attributes.



**Coding**

Let's start writing the smart contract. First, we’ll define the structure of the store contract state:

| pub struct AttributeStore {  admin: ActorId,  ft\_contract\_id: ActorId,  attributes: BTreeMap<AttributeId, (Metadata, Price)>,  owners: BTreeMap<TamagotchiId, BTreeSet<AttributeId>>, } |
| --- |

We’ll use *type alias* to improve the code readability:

| pub type AttributeId = u32; pub type Price = u128; pub type TamagotchiId = ActorId; |
| --- |

The Metadata for the attribute contains the following fields:

| pub struct **Metadata** {  *// the attribute title, for example: "Weapon"*  pub title: String,  *// description of the attribute*  pub description: String,  *// URL to associated media (here it should be an attribute picture)*  pub media: String, } |
| --- |

Let’s define the actions that the store contract must execute:

* The contract must create new attributes and sell them to the Tamagotchi contracts;
* The contract must receive messages from the Tamagotchi contracts.

Before implementing these functions, we’ll define the *contract store's store-io crate and write the* *lib.rs* file:

| **#![no\_std]** use gstd::{prelude::\*, ActorId};  pub type **AttributeId** = u32; pub type **Price** = u128; pub type **TamagotchiId** = ActorId; **#[derive(Encode, Decode)]** pub struct **Metadata** {  *// the attribute title, for example: "Weapon"*  pub title: String,  *// description of the attribute*  pub description: String,  *// URL to associated media (here it should be an attribute picture)*  pub media: String, }  **#[derive(Encode, Decode)]** pub enum **StoreAction** {  CreateAttribute {  attribute\_id: AttributeId,  metadata: Metadata,  price: Price  },  BuyAttribute {  attribute\_id: AttributeId,  } }  **#[derive(Encode, Decode)]** pub enum **StoreEvent** {  AttributeCreated {  attribute\_id: AttributeId,  },  AttributeSold {  success: bool,  }, } |
| --- |

The store contract will accept two types of messages: *CreateAttribute* and *BuyAttribute*. On successful message execution, it'll reply with *AttributeCreated* or *AttributeSold.*

We’ll then write the basic structure of the program as follows:

| **#![no\_std]** use gstd::{msg, prelude::\*, ActorId};  use store\_io::\*;  static mut STORE: Option<AttributeStore> = None;  pub struct **AttributeStore** {  admin: ActorId,  ft\_contract\_id: ActorId,  attributes: BTreeMap<AttributeId, (Metadata, Price)>  owners: BTreeMap<TamagotchiId, BTreeSet<AttributeId>) }  impl AttributeStore {  fn **create\_attribute**(&mut self, attribute\_id: AttributeId, metadata: &Metadata, price: Price) {}  async fn **buy\_attribute**(&mut self, attribute\_id: AttributeId) {}  }  **#[gstd::async\_main]** async fn **main**() {  let action: StoreAction = msg::load().expect("Unable to decode `StoreAction`");  let store: &mut AttributeStore = unsafe { STORE.get\_or\_insert(Default::default()) };  match action {  StoreAction::CreateAttribute {  attribute\_id,  metadata,  price  } => store.create\_attribute(attribute\_id, &metadata, price),  StoreAction::BuyAttribute { attribute\_id } => store.buy\_attribute(attribute\_id).await,  } }  **#[no\_mangle]** unsafe extern "C" fn **init**() {  let ft\_contract\_id: ActorId = msg::load().expect("Unable to decode `ActorId`);  let store = AttributeStore {  admin: msg::source(),  ft\_contract\_id,  ..Default::default()  };  STORE = Some(store); } |
| --- |

The buy\_attribute function is asynchronous since the store contract must send a message to the token contract and wait for a reply from it.

Now, let's implement the create\_attribute function. This function is straightforward and performs the following steps:

* Verifies that the account that sent the message is the contract admin.
* Ensures that an attribute with the indicated ID doesn't already exist.
* Creates a new attribute
* Sends a reply indicating the successful creation of the attribute.

| fn create\_attribute(&mut self, attribute\_id: AttributeId, metadata: &Metadata, price: Price) {  assert\_eq!(msg::source(), self.admin, "Only admin can add attributes");   if self  .attributes  .insert(attribute\_id, (metadata.clone(), price))  .is\_some()  {  panic!("Attribute with that ID already exists");  }   msg::reply(StoreEvent::AttributeCreated { attribute\_id }, 0)  .expect("Error in sending a reply `StoreEvent::AttributeCreated");  } |
| --- |

Next, let's dive into the implementation of the **buy\_attribute** function. As we discussed earlier, this function is responsible for initiating a token transfer from the Tamagotchi contract to the store contract, and it must track the transaction's ID in the fungible token contract. To achieve this, we will add a new field called **transaction\_id** to the store contract's state.

So, the store contract is responsible for tracking the transactions in the fungible token and has to consider the ID of the current transaction in it. Let’s add the field transaction\_id to the contract state:

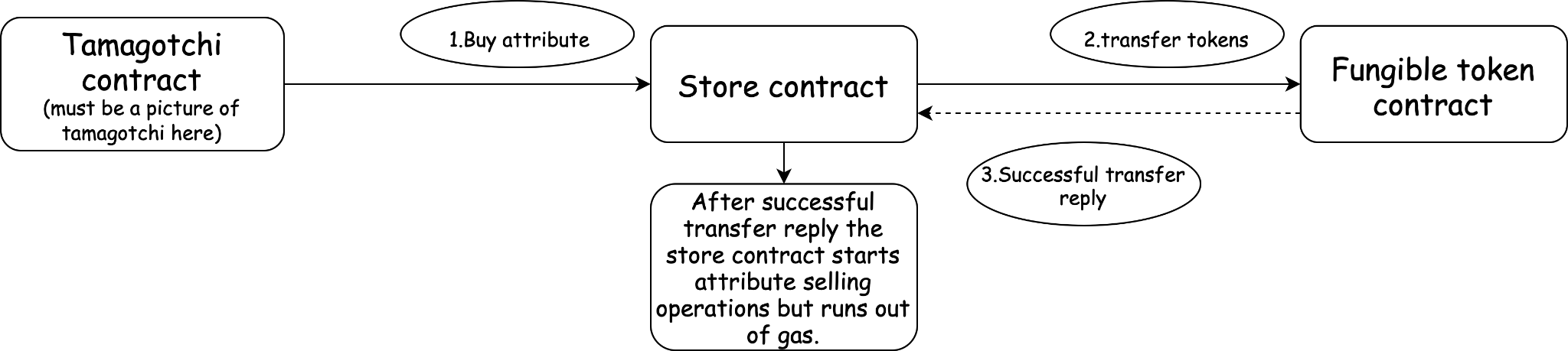
| pub struct **AttributeStore** {  ...  transaction\_id: TransactionId, } |
| --- |

This field will store the ID of the current transaction and will allow the store contract to track the status of the token transfer with ease. With this field in place, the buy\_attribute function can initiate the token transfer, track the transaction's ID, and wait for a reply from the fungible token contract to confirm the transfer's success.

And we also declare the type for transaction **id** in the **store-io** crate:

| pub type TransactionId = u64; |
| --- |

Next, let’s assume the following situations:



1. The Tamagotchi sends a message to the store contract to buy an attribute;
2. The store contract sends a message to the fungible token contract and receives a reply about the successful token transfer;
3. The store contract begins changing its state. It adds the indicated attribute to the Tamagotchi ownership but runs out of gas.

In such a scenario, the tokens were transferred to the store contracts but the Tamagotchi didn’t receive its attribute. To prevent this, it's important for the store contract to detect when a transaction is incomplete and continue its execution accordingly.

Let’s add another field to the *AttributeStore* struct:

| pub struct AttributeStore {  ...  transaction\_id: TransactionId,  transactions: BTreeMap<TamagotchiId, (TransactionId, AttributeId)>, } |
| --- |

When the store contract receives a purchase message from a Tamagotchi, it checks if the Tamagotchi is already involved in any incomplete transactions.

If the Tamagotchi has an incomplete transaction, the store contract retrieves the transaction number and attribute ID associated with the transaction, and resumes the transaction.

If the previous message wasn’t completed, the Tamagotchi has to send another identical message to complete the transaction. However, it's possible that the Tamagotchi sends multiple purchase messages and fails to notice that some messages did not go through.

To handle this, the store contract checks the attribute ID specified in the current message and compares it with the attribute ID stored in transactions. If the saved id is not equal to the indicated one, then the store contract asks the Tamagotchi to complete the previous transaction. Otherwise, it continues the pending transaction.

If the Tamagotchi has no pending transactions, then the store contract increments the *transaction\_id* and saves the transaction.

| async fn **buy\_attribute**(&mut self, attribute\_id: AttributeId) {  let (transaction\_id, attribute\_id) = if let Some((transaction\_id, prev\_attribute\_id)) =  self.transactions.get(&msg::source())  {  *// if `prev\_attribute\_id` is not equal to `attribute\_id` then it means that transaction didn`t completed*  *// we ask the Tamagotchi contract to complete the previous transaction*  if attribute\_id != \*prev\_attribute\_id {  msg::reply(  StoreEvent::CompletePrevTx {  attribute\_id: \*prev\_attribute\_id,  },  0,  )  .expect("Error in sending a reply `StoreEvent::CompletePrevTx`");  return;  }  (\*transaction\_id, \*prev\_attribute\_id)  } else {  let current\_transaction\_id = self.transaction\_id;  self.transaction\_id = self.transaction\_id.wrapping\_add(1);  self.transactions  .insert(msg::source(), (current\_transaction\_id, attribute\_id));  (current\_transaction\_id, attribute\_id)  };   let result = self.sell\_attribute(transaction\_id, attribute\_id).await;  self.transactions.remove(&msg::source());   msg::reply(StoreEvent::AttributeSold { success: result }, 0)  .expect("Error in sending a reply `StoreEvent::AttributeSold`");  } |
| --- |

Note that you have to add **CompletePrevTx** event to **StoreEvent** to ensure proper event tracking.

Let’s write the function for selling attributes. Selling attributes is similar to executing the NFT transfer. We’ll assign the attribute ID to the Tamagotchi contract.

First, we’ll write the function for the token transfer:

| async fn transfer\_tokens(  transaction\_id: TransactionId,  token\_address: &ActorId,  from: &ActorId,  to: &ActorId,  amount\_tokens: u128, ) -> Result<(), ()> {  let reply = msg::send\_for\_reply\_as::<\_, FTokenEvent>(  \*token\_address,  FTokenAction::Message {  transaction\_id,  payload: Action::Transfer {  sender: \*from,  recipient: \*to,  amount: amount\_tokens,  }  .encode(),  },  0,  )  .expect("Error in sending a message `FTokenAction::Message`")  .await;   match reply {  Ok(FTokenEvent::Ok) => Ok(()),  \_ => Err(()),  } } |
| --- |

We’ve sent a message to the token contract and handled its reply. The contract considers that the message to the token contract was successfully processed only if it received the *FTokenEvent::Ok*.

Now, we’re ready to write the function for selling attributes:

| async fn **sell\_attribute**(  &mut self,  transaction\_id: TransactionId,  attribute\_id: AttributeId,  ) -> bool {  let (\_, price) = self  .attributes  .get(&attribute\_id)  .expect("Can`t get attribute\_id");   if transfer\_tokens(  transaction\_id,  &self.ft\_contract\_id,  &msg::source(),  &exec::program\_id(),  \*price,  )  .await  .is\_ok()  {  self.owners  .entry(msg::source())  .and\_modify(|attributes| {  attributes.insert(attribute\_id);  })  .or\_insert\_with(|| [attribute\_id].into());  return true;  }  false  } |
| --- |

First, the contract receives the attribute price, then it calls the function *transfer\_tokens*. If the result of the token transfer is successful, it adds the attribute to the Tamagotchi contract.

Great! We’re done writing the contract logic.

Now, you should give your Tamagotchi the ability to buy attributes.

**What we have learnt:**

* Communicating with the fungible token contract;
* How to handle incomplete/imperfect transactions.

**Homework:**

* Give the tokens to your Tamagotchi contract (here must be the link to the fungible token contract deployed on the testnet);
* Add fields to the Tamagotchi contract that store the address of the fungible token contract;
* Add the ability to approve to transfer its token (and accordingly the field transaction\_id for communication with the fungible token contract);
* Add the function buy\_attribute to your Tamagotchi contract;
* Buy attributes for your Tamagotchi and see how it's transforming.

For the contract to be in accordance with the frontend, the metadata must be the following:

| pub struct ProgramMetadata;  impl Metadata for ProgramMetadata {  type Init = InOut<String, ()>;  type Handle = InOut<TmgAction, TmgEvent>;  type Reply = InOut<(), ()>;  type Others = InOut<(), ()>;  type Signal = ();  type State = Tamagotchi; }  *#[derive(Encode, Decode, TypeInfo)]* pub enum TmgAction {  Name,  Age,  Feed,  Play,  Sleep,  Transfer(ActorId),  Approve(ActorId),  RevokeApproval,  ApproveTokens {  account: ActorId,  amount: u128,  },  SetFTokenContract(ActorId),  BuyAttribute {  store\_id: ActorId,  attribute\_id: AttributeId,  }, }  *#[derive(Encode, Decode, TypeInfo)]* pub enum TmgEvent {  Name(String),  Age(u64),  Fed,  Entertained,  Slept,  Transfer(ActorId),  Approve(ActorId),  RevokeApproval,  ApproveTokens { account: ActorId, amount: u128 },  ApprovalError,  SetFTokenContract,  AttributeBought(AttributeId),  CompletePrevPurchase(AttributeId),  ErrorDuringPurchase, } |
| --- |