**Lesson 2: Building an Escrow Smart Contract**

**Lesson Summary**:

* Escrow smart contracts eliminate the need for traditional third-party intermediaries.
* Smart contracts are encoded in blockchain, making them secure, transparent, and tamper-proof.
* Participants learn to build an escrow smart contract using blockchain programming languages.
* Testing and deploying an escrow smart contract on a blockchain platform is an essential aspect of our lesson.

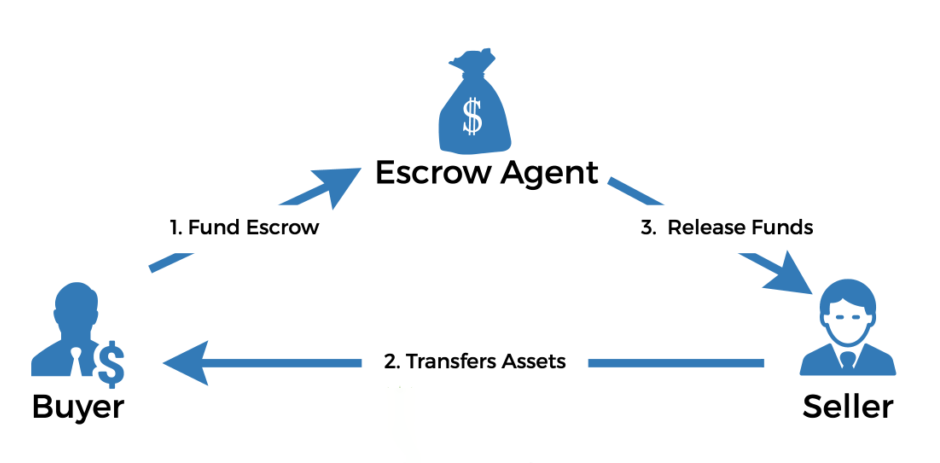
**Lesson Objective:**

By the end of the lesson, you will:

* Understand the concept of an escrow smart contract and how it works.
* Gain hands-on experience in building an escrow smart contract.
* Read program state using your own function.
* Learn how to test an escrow smart contract

**Let’s get started!**

An Escrow is an arrangement for a third party to temporarily hold the assets of a transaction. The assets are kept in the third party account and are only released once all agreement terms are met. Using an escrow account in a transaction adds safety for both parties.



Encoding an escrow smart contract into a blockchain enables the contract to be securely executed without traditional third parties.

**How an Escrow in Smart Contract Works**

1. The Buyer and Seller agree to the escrow terms. The Seller lists an item for sale, and the Buyer pays the price agreed upon.
2. The Seller delivers the product, and the funds are held in the escrow smart contract until all conditions are met and the Buyer confirms receipt of the product.
3. Once the Buyer approves the product, the funds will be automatically transferred to the Seller's digital wallet by the smart contract.

**Escrow Project Coding Practice**

Let’s create a new project with the following command:

| cargo new escrow --lib |
| --- |

We need to add the necessary dependencies to the Cargo.toml file and create a *build.rs* file (similar to the hello-world lesson).

Our program must store several states to correctly execute the logic. These states include the addresses of the buyer and seller, the product price, and the transaction state.

1. *AwaitingPayment*: Seller listed an item for sale but Buyer hasn’t sent funds yet;
2. *AwaitingDelivery*: Buyer transferred the funds to the smart contract, Seller sent the product;
3. *Closed*: The buyer confirmed the delivery and the Seller received the funds.

Let’s define these states in enum:

| enum **EscrowState** {  AwaitingPayment,  AwaitingDelivery,  Closed, } |
| --- |

Next, let’s define the structure that will store all necessary states:

| struct Escrow {  seller: ActorId,  buyer: ActorId,  price: u128,  state: EscrowState, } |
| --- |

We also need a global variable that will undergo changes during the contract execution. We'll use the `static mut` construct for this:

| static mut ESCROW: Option<Escrow> = None; |
| --- |

Until the program is initialized, the *ESCROW* value equals **None**. During initialization, we will fill the Escrow structure with information, and *ESCROW* will become **Some(Escrow)**.

Here’s the full code with minimal Gear smart contract structure l:

| **#![no\_std]** use gstd::{msg, ActorId, prelude::\*};   enum **EscrowState** {  AwaitingPayment,  AwaitingDelivery,  Closed, }  struct **Escrow** {  seller: ActorId,  buyer: ActorId,  price: u128,  state: EscrowState, }  static mut ESCROW: Option<Escrow> = None;   **#[no\_mangle]** unsafe extern "C" fn **handle** () {} **#[no\_mangle]** unsafe extern "C" fn **init** () {} |
| --- |

Build the project with the **cargo build --release** command and ensure everything works.

We’ll then describe and write the init function.

Let's define the InitEscrow message payload that will be sent during initialization. This structure needs to implement the **Encode** and **Decode** traits to be encoded and decoded, and also the **TypeInfo** trait for reading state.

| **#[derive(Encode, Decode, TypeInfo)]** pub struct **InitEscrow** {  pub seller: ActorId,  pub buyer: ActorId,  pub price: u128, } |
| --- |

In the **init** function, we'll define the Buyer's and Seller's addresses, as well as the product price. We'll load the message using **msg::load()** and decode it using the **InitEscrow** structure. Then, we'll create a new **Escrow** structure with the information and set the **state** to **EscrowState::AwaitingPayment**. Finally, we'll set **ESCROW** to **Some(escrow)**.

Let’s load the message in init function and define the contract state:

| **#[no\_mangle]** unsafe extern "C" fn **init** () {  let init\_config: InitEscrow = msg::load().expect("Error in decoding `InitEscrow`");  let escrow = Escrow {  seller: init\_config.seller,  buyer: init\_config.buyer,  price: init\_config.price,  state: EscrowState::AwaitingPayment,  };  ESCROW = Some(escrow); } |
| --- |

Now, we'll write the escrow contract logic. Our contract will handle the following messages:

* Message from Buyer with attached funds. The escrow contract checks that:
* The escrow state is *AwaitingPayment*;
* Sender’s address is equal to Buyer’s address;
* The attached funds equal the product price.

Then, the contract sets the escrow state to *AwaitingDelivery* and sends the reply about the successful fund deposit.

* Message from Buyer confirming the receipt of the goods. The escrow contract checks that:
* The escrow state is *AwaitingDelivery*;
* The Sender’s address is equal to the Buyer’s address.

Then the contract sets the escrow state to *Closed*, sends funds to the Seller, and sends the reply about successful escrow closure.

Great! Now, we need to declare the enums for incoming and outcoming messages, methods for Escrow structure, and implement the handle function.

| **#[derive(Encode, Decode, TypeInfo)]** pub enum **EscrowAction** {  Deposit,  ConfirmDelivery, }  **#[derive(Encode, Decode, TypeInfo)]** pub enum **EscrowEvent** {  FundsDeposited,  DeliveryConfirmed, }  impl Escrow {  fn **deposit**(&mut self) {}  fn **confirm\_delivery**(&mut self) {} } **#[no\_mangle]** unsafe extern "C" fn **handle**() {  let action: EscrowAction = msg::load().expect("Unable to decode `EscrowAction`");  let escrow: &mut Escrow = ESCROW.get\_or\_insert(Default::default());  match action {  EscrowAction::Deposit => escrow.deposit(),  EscrowAction::ConfirmDelivery => escrow.confirm\_delivery(),  } } |
| --- |

Note that we have to implement the *Default* trait for the Escrow structure. Let’s add the **#[derive(Default)]** above the Escrow structure and implement that trait for the *EscrowState* enum:

| impl Default for EscrowState {  fn default() -> Self {  Self::AwaitingPayment  } } |
| --- |

Let’s implement the deposit method:

* We’ll check that the contract state equals the *AwaitingDelivery* (For this, we have to add **#[derive(Debug,PartialEq, Eq)]** above the *EscrowState* enum):

| assert\_eq!(  self.state,  EscrowState::AwaitingPayment,  "State must be `AwaitingPayment"  ); |
| --- |

* Then check the sender account (To obtain *ActorId* of the account that sends the currently processing message we use the **source()** function from **msg** module in **gstd** library):

| assert\_eq!(  msg::source(),  self.buyer,  "The message sender must be a buyer"  )*;* |
| --- |

* And also check the attached funds (To get the value attached to the message being processed we use the **value()** function from **msg** module in **gstd** library):

| assert\_eq!(  msg::value(),  self.price,  "The attached value must be equal to set price"  )*;* |
| --- |

* Finally, we change the escrow state and send a reply message:

| self.state = EscrowState::AwaitingDelivery; msg::reply(EscrowEvent::FundsDeposited, 0)  .expect("Error in reply `EscrowEvent::FundsDeposited"); |
| --- |

**Reading program state using your own function**

In the previous lesson, we learnt how to read the full program state using state function in the contract. Additionally, it's possible to create your own library with functions to read the contract state.

Let’s extend the functionality of our escrow program by adding program metadata. We’ll start by creating a crate escrow-io in the directory of the escrow program:

| cargo new io --lib |
| --- |

The Cargo.toml file of that crate will contain the following:

| [package] name = "escrow-io" version = "0.1.0" edition = "2021"  [dependencies] gmeta = { git = "https://github.com/gear-tech/gear.git" } gstd = { git = "https://github.com/gear-tech/gear.git" } codec = { package = "parity-scale-codec", version = "3.1.2", default-features = false, features = ["derive", "full"] } scale-info = { version = "2.0.1", default-features = false, features = ["derive"] } |
| --- |

Now we can move InitEscrow, EscrowAction, EscrowEvent, EscrowState and Escrow to that crate and define the ProgramMetadata as follows:

| *#![no\_std]*  use gmeta::{InOut, Metadata}; use gstd::{prelude::\*, ActorId}; use scale\_info::TypeInfo;  pub struct ProgramMetadata;  impl Metadata for ProgramMetadata {  type Init = InOut<InitEscrow, ()>;  type Handle = InOut<EscrowAction, EscrowEvent>;  type Reply = InOut<(), ()>;  type Others = InOut<(), ()>;  type Signal = ();  type State = EscrowState; }  *#[derive(Encode, Decode, TypeInfo)]* pub struct InitEscrow {  pub seller: ActorId,  pub buyer: ActorId,  pub price: u128, }  *#[derive(Encode, Decode, TypeInfo)]* pub enum EscrowAction {  Deposit,  ConfirmDelivery, }  *#[derive(Encode, Decode, TypeInfo)]* pub enum EscrowEvent {  FundsDeposited,  DeliveryConfirmed, }  *#[derive(Debug, PartialEq, Eq, Encode, Decode, TypeInfo)]* pub enum EscrowState {  AwaitingPayment,  AwaitingDelivery,  Closed, }  impl Default for EscrowState {  fn default() -> Self {  Self::AwaitingPayment  } }  *#[derive(Default, Encode, Decode, TypeInfo)]* pub struct Escrow {  pub seller: ActorId,  pub buyer: ActorId,  pub price: u128,  pub state: EscrowState, } |
| --- |

To add a state function to the escrow-io crate, we include:

| **#[no\_mangle]** extern "C" fn **state**() {  let escrow = unsafe {  ESCROW.get\_or\_insert(Default::default())  };  msg::reply(escrow, 0).expect("Failed to share state"); } |
| --- |

Add dependencies to Cargo.toml of the escrow program:

| [package] name = "escrow" version = "0.1.0" edition = "2021"  [dependencies] gstd = { git = "https://github.com/gear-tech/gear.git", features = ["debug"] } codec = { package = "parity-scale-codec", version = "3.2.1", default-features = false } scale-info = { version = "2.2.0", default-features = false } escrow-io = { path = "io" }  [build-dependencies] gear-wasm-builder = { git = "https://github.com/gear-tech/gear.git"} escrow-io = { path = "io" }  [dev-dependencies] gtest = { git = "https://github.com/gear-tech/gear.git"} |
| --- |

We’ll change the build.rs file:

| fn main() {  gear\_wasm\_builder::build\_with\_metadata::<escrow\_io::ProgramMetadata>(); } |
| --- |

And create an independent crate for reading state:

| cargo new state --lib |
| --- |

The Cargo.toml of this crate will contain the following:

| [package] name = "escrow-state" version = "0.1.0" edition = "2018"  [dependencies] gmeta = { git = "https://github.com/gear-tech/gear.git", features = ["codegen"] } gstd = { git = "https://github.com/gear-tech/gear.git" } codec = { package = "parity-scale-codec", version = "3.1.2", default-features = false, features = ["derive", "full"] } scale-info = { version = "2.0.1", default-features = false, features = ["derive"] } escrow-io = { path = "../io" }  [build-dependencies] gear-wasm-builder = { git = "https://github.com/gear-tech/gear.git", features = ["metawasm"] } |
| --- |

In the lib.rs file, we should define metawasm trait as follows:

| **#![no\_std]** use gmeta::metawasm; use gstd::{prelude::\*, ActorId}; use escrow\_io::\*;  **#[metawasm]** pub trait **Metawasm** {  ... } |
| --- |

It’s also necessary to define the type of program state, which is the Escrow type in this case. We can execute this by adding `type State = Escrow;`:

| **#![no\_std]** use gmeta::metawasm; use gstd::{prelude::\*, ActorId}; use escrow\_io::\*;  **#[metawasm]** pub trait **Metawasm** {  type **State** = Escrow;  ...  } |
| --- |

Now that we've defined the trait and the state type, we can write any functions we want that concern the Escrow state. For example:

| *#![no\_std]* use gmeta::metawasm; use gstd::{prelude::\*, ActorId}; use escrow\_io::\*;  *#[metawasm]* pub trait Metawasm {  type State = Escrow;   fn seller(state: Self::State) -> ActorId {  state.seller  }   fn buyer(state: Self::State) -> ActorId {  state.buyer  }   fn escrow\_state(state: Self::State) -> EscrowState {  state.state  } } |
| --- |

Finally, we’ll create the build.rs file of the state as follows:

| fn main() {  gear\_wasm\_builder::build\_metawasm(); } |
| --- |

Once we've built the crate, we'll have a file called **escrow\_state.meta.wasm** that we can use in our UI applications to interact with the smart contract.

**Testing our Smart Contract Program**

Let’s test our method.

We’ll first create the tests directory and escrow\_test.rs file:

| mkdir tests touch escrow\_test.rs |
| --- |

We’ll import necessary structures from **gtest** library and escrow crate and define constants for *Buyer*, *Seller* and product price. Then, we’ll send an init message using the following code :

| use escrow::{InitEscrow, EscrowAction, EscrowEvent}; use gtest::{Log, Program, System}; const BUYER: u64 = 100; const SELLER: u64 = 101; const PRICE: u128 = 100\_000;  **#[test]** fn **deposit**() {  let sys = System::new();  sys.init\_logger();  let escrow = Program::current(&sys);  let res = escrow.send(  SELLER,  InitEscrow {  seller: SELLER.into(),  buyer: BUYER.into(),  price: PRICE,  },  );  assert!(res.log().is\_empty()); } |
| --- |

Next, we’ll send a message from the Buyer’s account using the send\_with\_value function instead of send function since we need to send a message with funds. However, in the test node, the account balance is zero, so we’ll have to change it:

| sys.mint\_to(BUYER, PRICE);  let res = escrow.send\_with\_value(BUYER, EscrowAction::Deposit, PRICE); let log = Log::builder()  .dest(BUYER)  .payload(EscrowEvent::FundsDeposited); assert!(res.contains(&log)); |
| --- |

To keep things organized, let's move the contract initialization into a separate function called **init\_escrow(sys: &System)**:

| fn init\_escrow(sys: &System) {  sys.init\_logger();  let escrow = Program::current(&sys);  let res = escrow.send(  SELLER,  InitEscrow {  seller: SELLER.into(),  buyer: BUYER.into(),  price: PRICE,  },  );  assert!(res.log().is\_empty()); } |
| --- |

We can use the *get\_program* function from the **gtest** library to get the program in the test function. As you remember from the first lesson, our program is initialized with the first id. So, the full code of the deposit test function is as follows:

| const ESCROW\_ID: u64 = 1;  **#[test]** fn **deposit**() {  let sys = System::new();  init\_escrow(&sys);   let escrow = sys.get\_program(ESCROW\_ID);   sys.mint\_to(BUYER, PRICE);   let res = escrow.send\_with\_value(BUYER, EscrowAction::Deposit, PRICE);  let log = Log::builder()  .dest(BUYER)  .payload(EscrowEvent::FundsDeposited);  assert!(res.contains(&log));   let escrow\_balance = sys.balance\_of(ESCROW\_ID);  assert\_eq!(escrow\_balance, PRICE); } |
| --- |

At the end of the test, we’ll also check that the funds are credited to the program using the **balance\_of** function.

It's crucial to test the correct contract execution and the failed cases. We have to check that the contract panics if:

* The message was sent from the wrong account;
* Buyer attached not enough funds;
* The escrow state is not *AwaitingPayment*.

So, let’s test all panics in the **deposit** function:

| **#[test]** fn **deposit\_failures**() {  let sys = System::new();  init\_escrow(&sys);   let escrow = sys.get\_program(ESCROW\_ID);   sys.mint\_to(BUYER, 2\*PRICE);  *// must fail since BUYER attaches not enough value*  let res = escrow.send\_with\_value(BUYER, EscrowAction::Deposit, 2\*PRICE - 500);  assert!(res.main\_failed());   *// must fail since the message sender is not BUYER*  let res = escrow.send(SELLER, EscrowAction::Deposit);  assert!(res.main\_failed());   *// successful deposit*  let res = escrow.send\_with\_value(BUYER, EscrowAction::Deposit, PRICE);  assert!(!res.main\_failed());   *// must fail since the state must be `AwaitingPayment`*  let res = escrow.send\_with\_value(BUYER, EscrowAction::Deposit, PRICE);  assert!(res.main\_failed()); } |
| --- |

Great, we have written half of our program. Now it's time for you to code.

**Assignment:**

1. Implement the **confirm\_delivery** function. The function should:

* Check that **msg::source()** is a buyer;
* Check that the escrow state is *AwaitingDelivery*;
* Sends the funds to the seller (use **msg::send()** function);
* Set the escrow state to *Closed*;
* Send a reply message about successful delivery confirmation.

1. Write tests for the written function:

* **confirm\_delivery** test that tests the successful contract execution;

Note that the contract sends a message with value to a seller and the user messages are stored in their individual **mailbox**. To get the value from these messages, it's necessary to claim the value from the mailbox.

In **gtest,** you can use the function claim\_value\_from\_mailbox. After claiming the value, check the seller's balance and make sure that funds were transferred to his account.

* **confirm\_delivery\_failures** test that tests all panics in the escrow contract.

1. Next, we return to the Tamagotchi contract you started writing in the previous lesson:

Let’s expand the Tamagochi state by adding the following field to its structure:

* The Tamagotchi owner (it can be an account that initializes the Tamagotchi contract);
* Mood: Fed (from 1 to 10000), Happy (from 1 to 10000) and Rested(from 1 to 10000). These values must be set to non-zero when initializing the Tamagotchi contract. Also, you should define the following constants:

**HUNGER\_PER\_BLOCK** = 1: how much Tamagotchi becomes hungry for the block;

**ENERGY\_PER\_BLOCK** = 2 - how much Tamagotchi loses energy per block;

**BOREDOM\_PER\_BLOCK** = 2 - how bored Tamagotchigetsper block;

**FILL\_PER\_SLEEP** = 1000 - how much energy Tamagotchi gets per sleep;

* **FILL\_PER\_FEED** = 1000 - how much Tamagotchi becomes full during feeding;
* **FILL\_PER\_ENTERTAINMENT** = 1000 - how much Tamagotchi becomes happy during feeding;
* The Tamagotchi also has to accept messages: **Sleep**, **Feed** and **Play**;
* Think of logic for calculating the levels of Fed, Happy and Rested. You need to take into account the numbers of blocks in which the Tamagotchi last ate, had fun or slept. For this you can use the function *block\_timestamp()* from the module *exec* of *gstd* library.

1. Now upload your contract to the blockchain and run the frontend application. Choose the second lesson.

Now you can feed your Tamagotchi, play with it and send it to sleep.

The metadata must meet the following requirements to ensure the contract aligns with the frontend:

| 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, }  *#[derive(Encode, Decode, TypeInfo)]* pub enum TmgEvent {  Name(String),  Age(u64),  Fed,  Entertained,  Slept, }  *#[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, } |
| --- |

Please attach a link to the repo with your Tamagotchi contract.