**Lesson 1: Creating the First Smart Contract Program - Hello World**

**Lesson Summary:**

* We will create our first smart contract program called "Hello World."
* Learn how to initialize a program and define the entry point for message processing.
* Add complexity to the program by restricting the type of messages it can receive and creating different greetings for different accounts, depending on the message.
* Explore testing smart contracts to ensure they work as intended.
* Learn how to read the program's state to retrieve important information.

**Lesson Objectives:**

By the end of the lesson, we will have:

* Defined the steps involved in creating a simple smart contract program named "Hello World."
* Explained the process of initializing a smart contract and defining the entry point for message processing.
* Identified different ways of restricting the type of messages a smart contract can receive, and how to create different greetings for different accounts based on the message received.
* Demonstrated how to test a smart contract program to ensure it functions correctly.
* Described the process of reading a smart contract's state to retrieve important information.
* Summarized the key concepts and building blocks required to develop functional smart contract programs.

**Let’s get started Implementing our First Program!**

Let’s implement the program that’ll send a hello message in response to any received messages.

To get started, we'll create a new project using the following command:

| cargo new hello-world --lib |
| --- |

This will create a directory structure for our project with the following files:

| └── hello-world  ├── Cargo.toml  └── src  └── lib.rs |
| --- |

Next, we need to add the necessary dependencies to our Cargo.toml file. We'll use

* gstd - a standard library for smart contracts on Gear
* gtest - a library for testing smart contracts
* gear-wasm-builder - a helping module that aids in building programs using Gear.

| [package] name = "hello-world" version = "0.1.0" edition = "2021"  [dependencies] gstd = { git = "https://github.com/gear-tech/gear.git", features = ["debug"] }  [build-dependencies] gear-wasm-builder = { git = "https://github.com/gear-tech/gear.git" }  [dev-dependencies] gtest = { git = "https://github.com/gear-tech/gear.git" } |
| --- |

Now, let's write the minimum structure of our Gear program in the lib.rs file. The handle function is the program's entry point. It will be called every time the program receives an incoming message.

| #![no\_std] use gstd::{msg, prelude::\*};  #[no\_mangle] unsafe extern "C" fn handle() {} |
| --- |

To build our program, we’ll create *build.rs* file with the following code:

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

We can now run the following command to build our project:

| cargo build *--*release |
| --- |

gstd::msg is the messaging module from the gstd library, allowing users to process incoming messages, obtain the necessary information about the sender or the message content, and send replies or new messages to other actors (link to gstd).

We'll use the reply function that sends a new message as a reply to the message currently being processed:

| #![no\_std] use gstd::{msg, prelude::\*};  #[no\_mangle] unsafe extern "C" fn handle() {  msg::reply(String::from("Hello"), 0).expect("Error in sending a reply message"); } |
| --- |

Let's build our project:

| cargo build *--*release |
| --- |

If everything goes well, your working directory should now have a target directory that resembles this:

| target  ├── release  │ └── ...  └── wasm32-unknown-unknown  └── release  ├── ...  ├── hello\_world.wasm <---- this is our built .wasm file  ├── hello\_world.opt.wasm <---- this is optimized .wasm file  └── hello\_world.meta.wasm <---- this is meta .wasm file |
| --- |

The target/wasm32-unknown-unknown/release directory should contain three WASM binaries:

* hello\_world.wasm is the output WASM binary built from source files;
* hello\_world.opt.wasm is the optimized WASM aimed to be uploaded to the blockchain;
* hello\_world.meta.wasm is the WASM containing meta information needed to interact with the program.

Before we upload our program to the blockchain, we need to understand the metadata and \*.meta.wasm file's use case. In the context of Gear programs, metadata facilitates the interaction between the client-side (JavaScript) and the program (Rust).

Metadata can be used as a message payload description for external tools and applications that interact with Gear programs in the network. It is stored in a separate \*.meta.wasm file.

The Gear `metadata!` macro exports functions from Rust with IO data specified by the user in the macro. In our example, we only need to declare a type for the handle output message:

| gstd::metadata! {  title: "Hello world contract",  handle:  output: String, } |
| --- |

We've learned how to create a simple smart contract program that responds with a "Hello" message to any incoming message. Let's test our program.

**Testing smart contract with gtest library**

Testing smart contracts is an important aspect of developing decentralized applications. We’ll use the Gear [gtest](https://github.com/gear-tech/gear/tree/master/gtest) library for our program’s logic testing.

To get started, let's create a new directory called "tests" at the top level of our project directory, next to the "src" director. In that directory, we’ll create a file hello\_world\_test.rs where we’ll write tests for our contract.

| mkdir tests touch hello\_world\_test.rs |
| --- |

In our test file, we’ll need to import the necessary modules from the gtest library, which are import *Log*, *Program* and *System*. We’ll also define a test function:

| use gtest::{Log, Program, System};  #[test] fn hello\_test() {} |
| --- |

Before testing our smart contract, we need to initialize the environment for running programs. We can do this using the System module from gtest. The system emulates the node behaviour:

| let sys = System::new(); |
| --- |

Next, we need to initialize our program. We can do this using the Program module from gtest. There are two ways to initialize a program: from a file or the current program:

To initialize a program from a file:

| let program = Program::from\_file(&sys,  "./target/wasm32-unknown-unknown/release/hello\_world.wasm"); |
| --- |

To initialize a program from the current program:

| let program = Program::current(&sys); |
| --- |

The uploaded program has its own id. You can specify the program id manually using the from\_file\_with\_id constructor. If you don't specify the program id, the id of the first initialized program will be 0x01000…, and the next program initialized without an id specification will have an id of 0x02000... and so on.

In the next step, we’ll send messages to our program.

* To send a message to the program, call one of two program functions: send() or send\_bytes(). The difference between them is similar to gstd functions msg::send and msg::send\_bytes.
* The first argument in these functions is a sender id, the second one is a message payload.
* The sender id can be specified as hex, array ([u8, 32]), string or u64. However, you can’t send a message from the id already taken by the program!
* The first message to the initialized program structure is always the init message even if the program does not have the init function. In our case, it can be any message. But let’s add the init function to our program and monitor if that message reaches the program:

| #![no\_std] use gstd::{msg, prelude::\*, debug};  #[no\_mangle] unsafe extern "C" fn handle() {  msg::reply("Hello", 0).expect("Error in sending a reply message"); }  #[no\_mangle] unsafe extern "C" fn init() {  let init\_message: String = msg::load().expect("Can't load init message");  debug!("Program was initialized with message {:?}", init\_message); } |
| --- |

In our test function, we can send a message to the program using the `send()` function:

| #[test] fn hello\_test() {  let sys = System::new();  sys.init\_logger();  let program = Program::current(&sys);  program.send(2, String::from("INIT MESSAGE")); } |
| --- |

Note that we added sys.init\_logger() to initialize printing logs into stdout, and we sent a message from the user with id 2 (id 2 transforms to 0x020000.. ActorId).

We can then run our test using the following command:

| cargo test *--*release |
| --- |

If everything is working correctly, we should see the debug message in our console:

| [DEBUG hello\_test] Program was initialized with message "INIT MESSAGE" test hello\_test ... ok |
| --- |

Sending functions in the gtest library will return RunResult structure. It contains the final result of the processing message and others, which were created during the execution.

For example, we can check the init message processing result. We can do this by ensuring that the log is empty and that the program does not reply or send any messages. To do this, we can use the assert!(res.log().is\_empty()) command.

* Contains empty log (the program doesn’t reply and does not send any messages);

| assert!(res.log().is\_empty())*;* |
| --- |

* Was successful:

| assert!(!res.main\_failed()); |
| --- |

Once we have confirmed that the initialization message was successful, the next messages will be processed through the handle function. We can test this by sending the next message using the program.send(2, String::from("HANDLE MESSAGE")) command.

| let res = program.send(2, String::from("HANDLE MESSAGE")); |
| --- |

Here, we should check that the program replied with the expected hello message. To do this, we can use the Log structure from the gtest lib and build the log we are expecting to receive. Specifically, we can use the Log::builder().dest(2).payload(String::from("Hello")) command to create the expected log.

After creating the expected log, we can then check if the received log contains the expected log. We can do this by using the assert!(res.contains(&expected\_log)) command.

| let expected\_log = Log::builder().dest(2).payload(String::from("Hello")); assert!(res.contains(&expected\_log)); |
| --- |

As you might guess, dest is the actor to which the program sends a message and payload is the content of that message.

Run the test and make sure that everything is fine.

**Lesson 1 Quiz: Testing Your Understanding**

1. Consider the following program:

| #![no\_std] use gstd::{msg, prelude::\*, debug};  #[no\_mangle] unsafe extern "C" fn handle() {  msg::reply("Hello", 0).expect("Error in sending a reply message"); } |
| --- |

We’ll test the program using the following test suite:

| #[test] fn hello\_test() {  let sys = System::new();  let program = Program::current(&sys);  let res = program.send(2, String::from("Hello"));  let expected\_log = Log::builder().dest(2);  assert!(res.contains(&expected\_log)); } |
| --- |

What will be the result of running that test?

* Everything will be fine: the test will pass;
* The test will fail because the first message to the initialized program structure is always the init message.
* The test will fail because the expected\_log does not contain the payload with “Hello”.

1. What is the mistake in writing that test?

| #[test] fn hello\_test() {  let sys = System::new();  let program\_1 = Program::current(&sys);  let program\_2 = Program::current(&sys);  let res = program\_2.send(2, String::from("Hello"));  assert!(res.log().is\_empty()); } |
| --- |

- The first message should be sent to the first initialized program;

* You can not initialize two identical programs;
* The message to the program\_2 is sent from the address that is already occupied by the program.

1. What returns fn send() from the gtest library?

* It returns the RunResult structure that contains the result of program execution;
* It returns the Log structure that contains the information about the source, destination, and payload of messages that were sent during the program execution;
* It returns the message that was sent back to the sender.

**Advanced Hello World Program Concepts**

Let’s add more functionality to our program by introducing two new messages: SendHelloTo and SendHelloReply.

Our program will receive 2 messages:

* SendHelloTo: having received this message, the program will send “hello” to the specified address;
* SendHelloReply: the program replies with a “hello” message to the account that sent the current message.

As we saw from the previous lesson, we’ll have to decode the message the program received. We’ll define an enum InputMessages that will be used to decode the incoming message.

| #[derive(Encode, Decode, TypeInfo)] enum InputMessages {  SendHelloTo(ActorId),  SendHelloReply, } |
| --- |

The SendHelloTo variant includes an ActorId field where the program will send the hello message.

We also need to add derive macros #[derive(Encode, Decode, TypeInfo)] to the enum for encoding and decoding in messages, and add appropriate dependencies to the Cargo.toml file:

| 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"] } |
| --- |

To initialize our program, we’ll define a static mutable variable GREETING as an Option<String>.

| static mut GREETING: Option<String> = None; |
| --- |

Until the program is initialized, the GREETING equals **None**. After the initialization, the GREETING will become **Some(String)**.

| **#[no\_mangle]** unsafe extern "C" fn **init**() {  let greeting = String::from\_utf8(msg::load\_bytes().expect("Can't load init message"))  .expect("Invalid message");  debug!("Program was initialized with message {:?}", greeting);  GREETING = Some(greeting); } |
| --- |

Next, we’ll decode the incoming message in the handle function and define the message the program received:

| **#[no\_mangle]** unsafe extern "C" fn **handle**() {  let input\_message: InputMessages = msg::load().expect("Error in loading InputMessages");  let greeting = GREETING.get\_or\_insert(Default::default());  match input\_message {  InputMessages::SendHelloTo(account) => {  debug!("Message: SendHelloTo {:?}", account);  msg::send(account, greeting, 0)  .expect("Error in sending Hello message to account");  }  InputMessages::SendHelloReply => {  debug!("Message: SendHelloReply");  msg::reply(greeting, 0).expect("Error in sending reply");  }  } } |
| --- |

When the program receives SendHelloTo message, it sends a hello message to the specified account through the send function. On the other hand, when the contract receives a SendHelloReply message, it replies with a greeting message.

**Testing the Updated Smart Contract**

First, we’ll test the SendHelloTo message. We define the account id that will receive that message and check that in the result log there is a message assigned to that account.

| use gtest::{Log, Program, System}; use hello\_world::InputMessages;  #[test] fn hello\_test() {  let sys = System::new();  sys.init\_logger();  let program = Program::current(&sys);  let res = program.send(2, String::from("INIT MESSAGE"));  assert!(!res.main\_failed());  assert!(res.log().is\_empty());   *//* test `SendHelloTo`  let hello\_recipient: u64 = 4;  let res = program.send(2, InputMessages::SendHelloTo(hello\_recipient.into()));  let expected\_log = Log::builder()  .dest(hello\_recipient)  .payload(String::from("Hello"));  assert!(res.contains(&expected\_log)) |
| --- |

**Understanding the program metadata and state**

Metadata is a kind of interface map that helps transform a set of bytes into an understandable structure. It determines how all incoming and outgoing data will be encoded/decoded.

Metadata allows dApp’s parts - the smart contract and the client side (JavaScript), to understand each other and exchange data.

To describe the metadata interface we use `gmeta` crate:

| use gmeta::{InOut, Metadata}; pub struct ProgramMetadata; impl Metadata for ProgramMetadata {  type Init = InOut<MessageInitIn, MessageInitOut>;  type Handle = InOut<MessageIn, MessageOut>;  type Others = InOut<MessageAsyncIn, Option<u8>>;  type Reply = InOut<String, Vec<u16>>;  type Signal = ();  type State = Vec<u128>; } |
| --- |

where:

* *Init* - describes incoming/outgoing types for `init()` function.
* *Handle* - describes incoming/outgoing types for `handle()` function.
* *Others* - describes incoming/outgoing types for `main()` function in case of asynchronous interaction.
* *Reply* - describes incoming/outgoing types of messages performed using the `handle\_reply` function.
* *Signal* - describes only the outgoing type from the program while processing the system signal.
* *State* - describes the types for the queried State

It is necessary to describe all the types. If any of the endpoints is missing in your program, you can use () instead.

Let's define metadata for our example. We’ll create a crate

hello-world-io in the directory of our hello-world program:

| cargo new hello-world-io --lib |
| --- |

The Cargo.toml of this crate:

| [package] name = "hello-world-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"] } |
| --- |

And in the lib.rs file, we’ll define an incoming message for the init function, and the incoming and outcoming messages for the handle function:

| *#![no\_std]*  use codec::{Decode, Encode}; use gmeta::{InOut, Metadata}; use gstd::{prelude::\*, ActorId}; use scale\_info::TypeInfo; pub struct ProgramMetadata;  impl Metadata for ProgramMetadata {  type Init = InOut<String, ()>;  type Handle = InOut<InputMessages, String>;  type Reply = InOut<(), ()>;  type Others = InOut<(), ()>;  type Signal = ();  type State = String; }  *#[derive(Encode, Decode, TypeInfo)]* pub enum InputMessages {  SendHelloTo(ActorId),  SendHelloReply, } |
| --- |

The input for init function is a *String*. The input for the handle function is an enum *InputMessage*, and accordingly, the output is *String*. The program state is also String (It is a set greeting).

It is possible to read the program state using the state function. Reading State is a free function and does not require gas costs. Let’s define this function in lib.rs file of the hello-world program:

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

To make it possible to verify metadata for a program, we’ll use the metahash() function:

| **#[no\_mangle]** *// It returns the Hash of metadata.* *// .metahash is generating automatically while you are using build.rs* extern "C" fn **metahash**() {  let metahash: [u8; 32] = include!("../.metahash");  msg::reply(metahash, 0).expect("Failed to share metahash"); } |
| --- |

It’s necessary to add the hello-world-io crate to build-dependencies in *Cargo.toml* in the hello-world program:

| [package] name = "hello-world" version = "0.1.0" edition = "2021" ... [build-dependencies] gear-wasm-builder = { git = "https://github.com/gear-tech/gear.git" } hello-world-io = { path = "hello-world-io" } |
| --- |

We also need to change the build.rs file using the following code:

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

After building the program, a meta.txt file will be generated as a result of the smart-contract compilation. This metadata file can be used in UI applications that will interact with this smart contract.

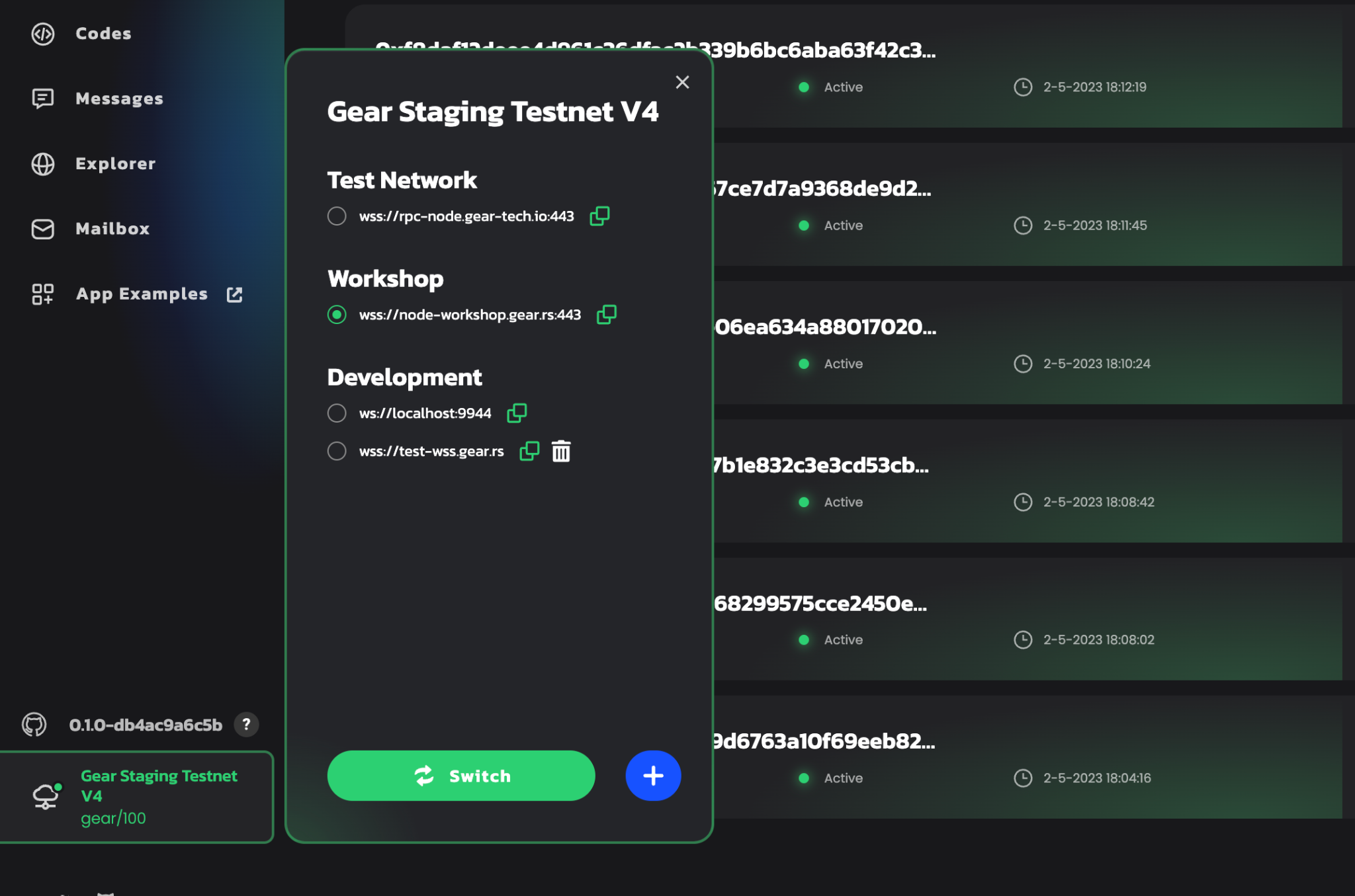
**Uploading the program to the blockchain**

The easiest way to upload the program is to use the “Upload program” option in the Gear Idea portal - [idea.gear-tech.io](https://idea.gear-tech.io/).

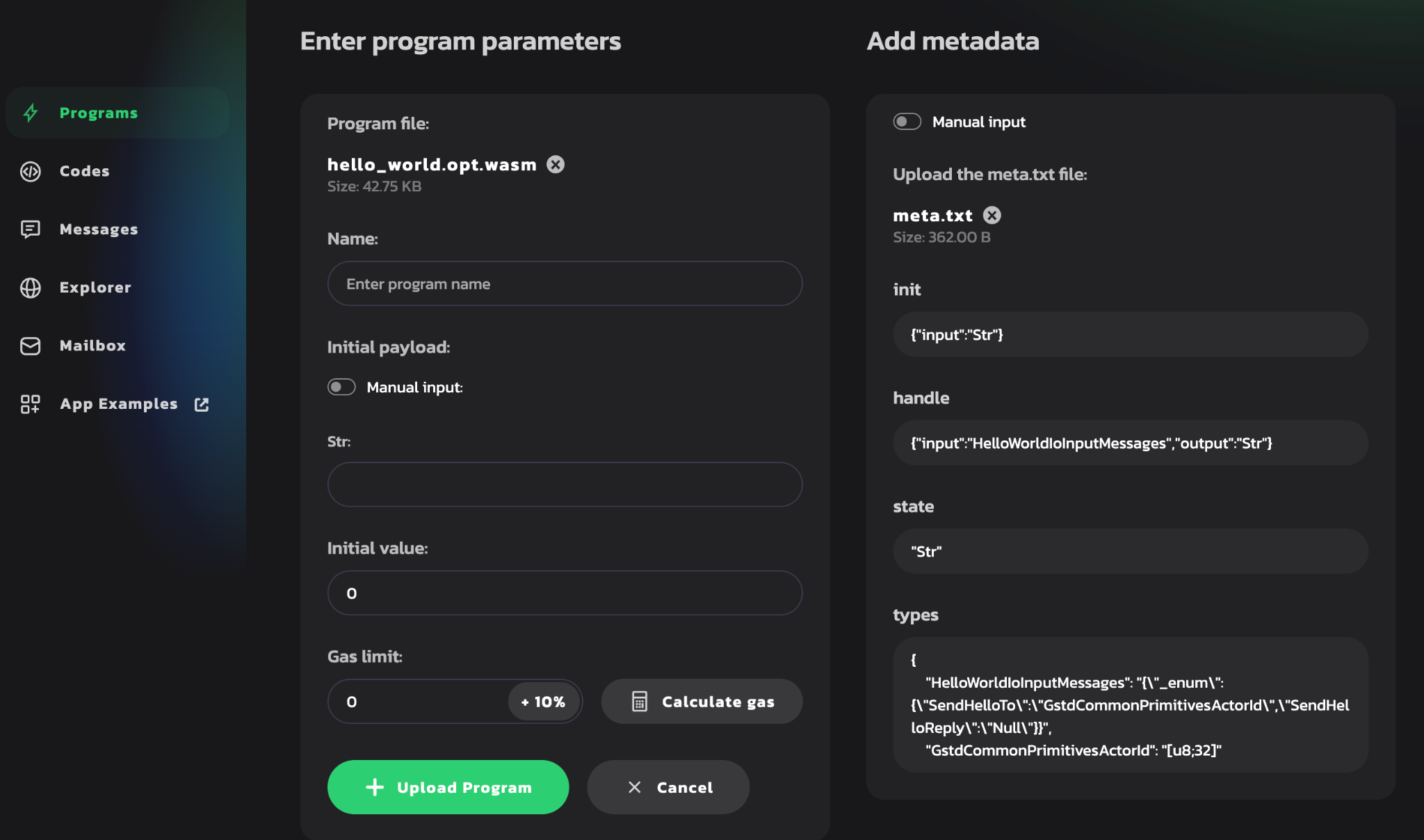
First, you need to create an account and connect to Gear Idea. Follow the instructions provided at <https://wiki.gear-tech.io/docs/idea/account/create-account> to create your account.

Once you've logged in, you can select the network on which you want to upload the program by clicking on the gear icon in the lower left corner of the screen. For the Gear Academy workshop, select the workshop node (wss://node-workshop.gear.rs:443) and click the "Switch" button.

Select the workshop node and click on the ***Switch*** button:

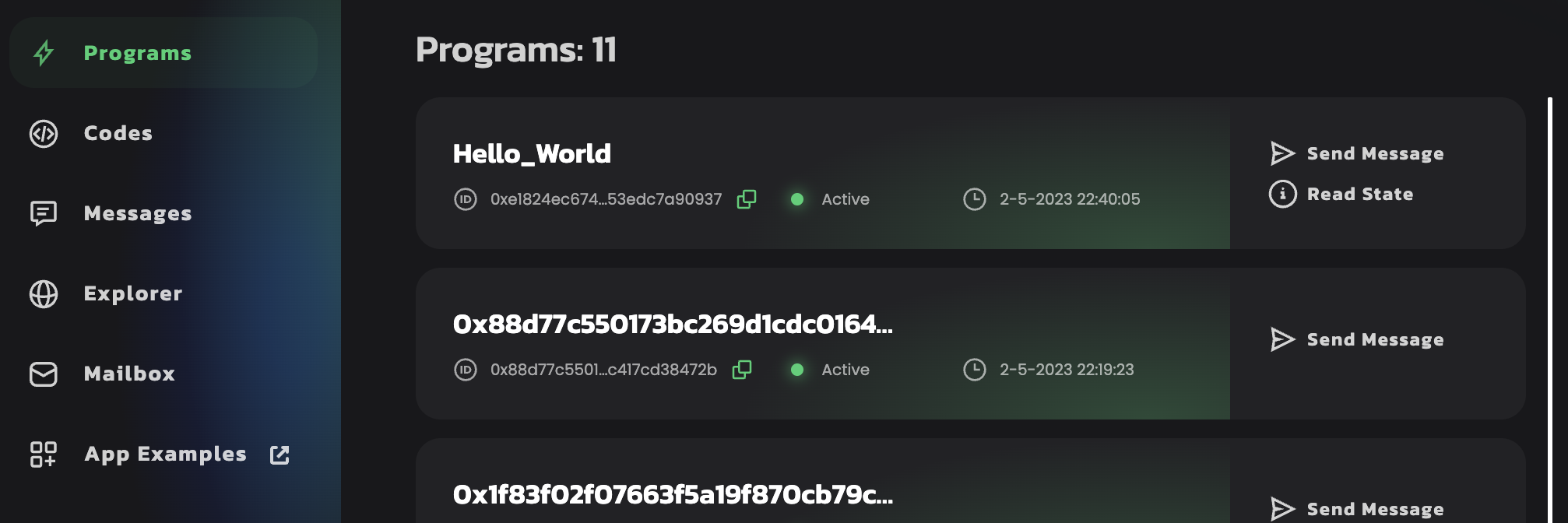


Next, select "Programs" in the left column and upload the hello.opt.wasm file along with its metadata (meta.txt file).

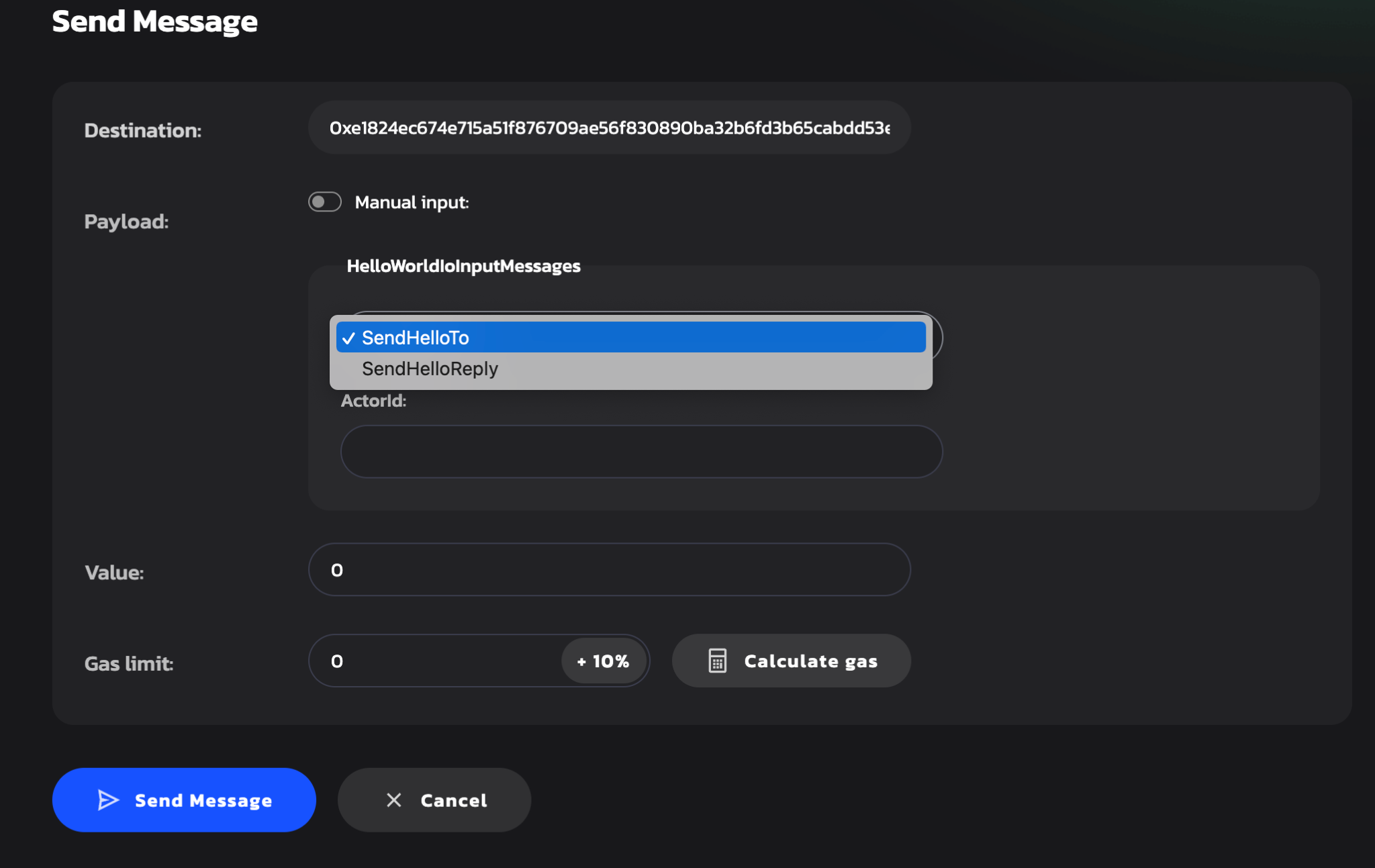


Give your program a name, enter an incoming greeting message and click on the ***Upload Program*** button.

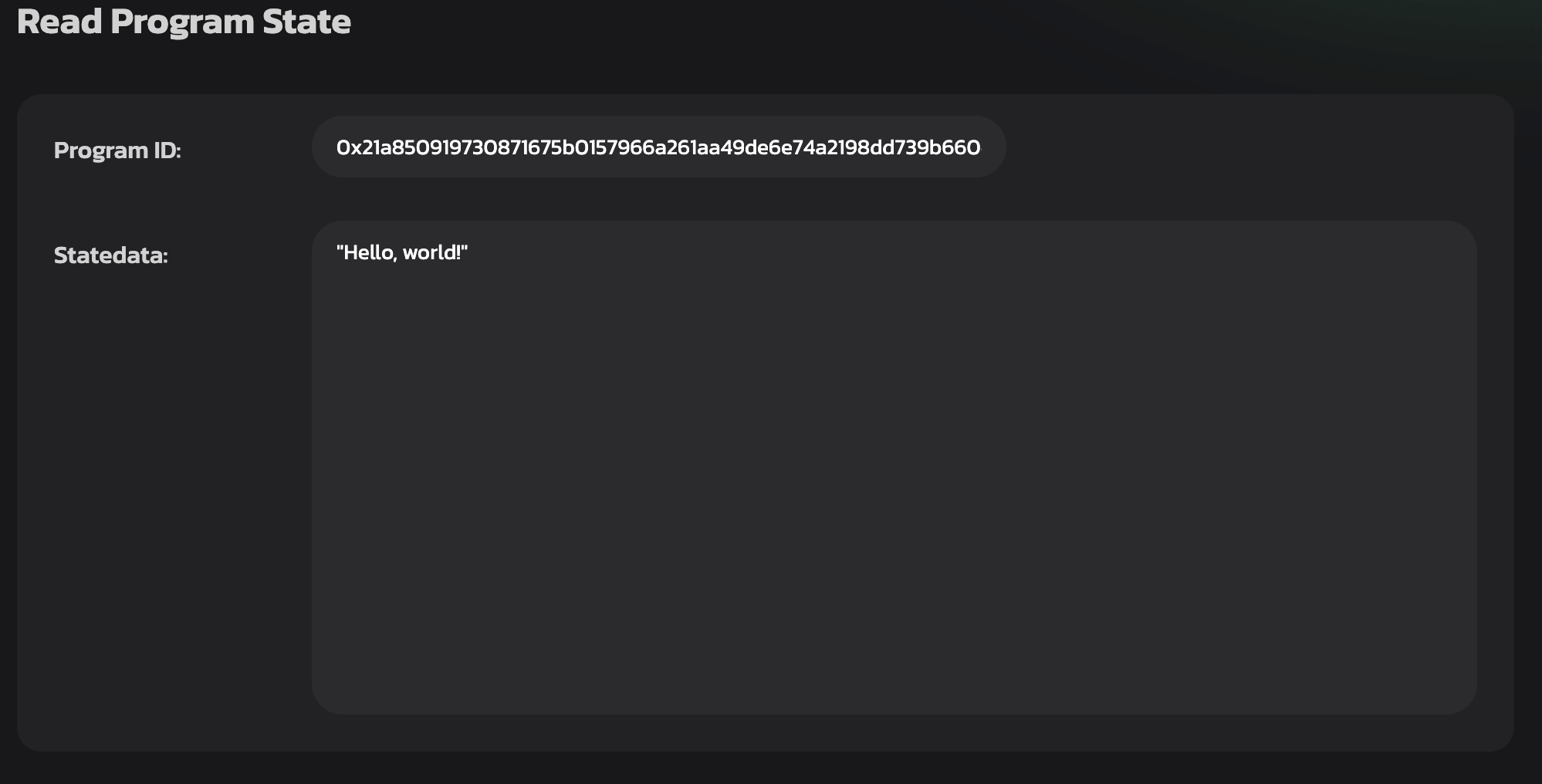
If the program has successfully uploaded, you will see it in the program.



You can now send messages to your program:



You can also read the program state (It’s our greeting string that was set during the program initialization).



**Assignment:**

Let's write a smart contract for a Tamagotchi game:

* Create a smart contract Tamagotchi, that will store Tamagotchi’s name and date of birth. he state of your contract should be defined as follows:

| **#[derive(Default, Encode, Decode, TypeInfo)]** pub struct **Tamagotchi** {  name: String,  date\_of\_birth: u64, } |
| --- |

* During initialization, set the name and date of birth of the Tamagotchi and send a reply confirming successful initialization.
* Your Tamagochi program should accept the following messages:
  + Name - the program answers the name of the Tamagochi;
  + Age - the program answers about the age of the Tamagochi.
* Add the state function to your program.
* Upload your contract to the workshop node at the <https://idea.gear-tech.io/>.

To connect your Tamagotchi contract to the frontend application, you need to ensure that the metadata is as follows:

| 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, }  *#[derive(Encode, Decode, TypeInfo)]* pub enum TmgEvent {  Name(String),  Age(u64), }  *#[derive(Default, Encode, Decode, TypeInfo)]* pub struct Tamagotchi {  name: String,  date\_of\_birth: u64, |
| --- |

}

Clone the repository: <https://github.com/gear-dapps/smart-contract-academy>

In the frontend directory, run the following commands:

| yarn yarn start |
| --- |

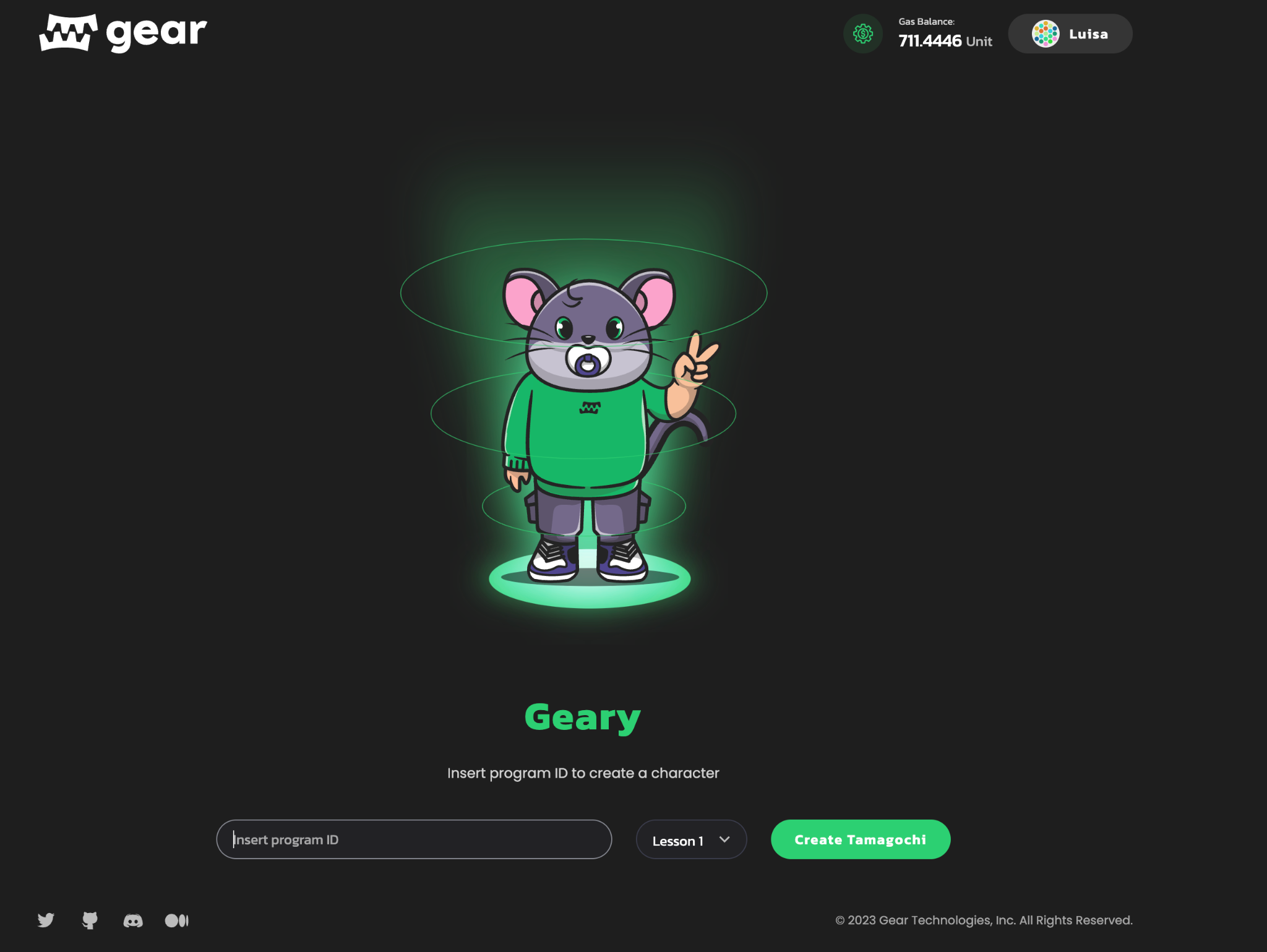
The .env.local file contains the following:

| REACT\_APP\_NODE\_ADDRESS=wss://test-wss.gear.rs |
| --- |

It means that the application is running on the testnet node. You can also run a local node, upload a Tamagotchi contract, and work with contracts on a local node by indicating:

| REACT\_APP\_NODE\_ADDRESS=ws://localhost:9944 |
| --- |

After running the ***yarn start***,command you will see the following window:



Select ***Lesson 1***, paste your Tamagotchi address and you’ll see your Tamagotchi!

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