Callbacks

NEAR Protocol is a sharded, proof-of-stake blockchain that behaves differently than proof-of-work blockchains. When interacting with a native Rust (compiled to Wasm) smart contract, cross-contract calls are asynchronous. Callbacks are used to either get the result of a cross-contract call or tell if a cross-contract call has succeeded or failed.

There are two techniques to write cross-contract callshigh-level andlow-level . This document will mostly focus on the high-level approach. There are two examples in the Rust SDK repository that demonstrate these, as linked above. Note that these examples use cross-contract calls "to itself." We'll show two examples demonstrating the high-level approach.

Calculator example

There is a helper macro that allows you to make cross-contract calls with the syntax#[ext_contract(...)] . It takes a Rust Trait and converts it to a module with static methods. Each of these static methods takes positional arguments defined by the Trait, then thereceiver_id , the attached deposit and the amount of gas and returns a newPromise .

info If the function returns the promise, then it will delegate the return value and status of transaction execution, but if you return a unit type (() ,void ,nothing), then thePromise result will not influence the transaction status. For example, let's define a calculator contract Trait:

[ext_contract(ext_calculator)]

trait

Calculator

{ fn

mult (& self , a :

U64 , b :

U64)

->

U128 ;

fn

sum (& self , a :

U128 , b :

U128)

->

U128 ; } It's equivalent to the following code:

mod

ext_calculator

{ pub

fn

mult (a :

U64 , b :

U64 , receiver_id :

& AccountId , deposit :

Balance , gas :

Gas)

->

Promise

{ Promise :: new (receiver_id . clone ( )) . function_call (b"mult" , json! ({

"a" : a ,

"b" : b }) . to_string ( ) . as_bytes ( ) , deposit , gas ,) }

pub

fn

sum (a :

U128 , b :

U128 , receiver_id :

& AccountId , deposit :

Balance , gas :

Gas)

->

Promise

{ // ... } } Let's assume the calculator is deployed oncalc.near , we can use the following:

[near_bindgen]

impl

Contract

{ pub

fn

sum_a_b (& mut

self , a :

U128 , b :

U128)

->

Promise

{ let calculator_account_id :

AccountId

=

"calc.near" . parse ( ) . unwrap ( ) ; // Call the method sum on the calculator contract. // Any unused GAS will be attached since the default GAS weight is 1. // Attached deposit is defaulted to 0. ext_calculator :: ext (calculator_account_id) . sum (a , b) } }

Allowlist example

Next we'll look at a simple cross-contract call that is made to an allowlist smart contract, returning whether an account is in the list or not.

The common pattern with cross-contract calls is to call a method on an external smart contract, use.then syntax to specify a callback, and then retrieve the result or status of the promise. The callback will typically live inside the same, calling smart

contract. There's a special macro used for the callback function, which is#[private] . We'll see this pattern in the example below.

The following example demonstrates two common approaches to callbacks using the high-level cross-contract approach. When writing high-level cross-contract calls, specialtraits are set up as interfaces for the smart contract being called.

[ext_contract(ext_allowlist)]

pub

trait

ExtAllowlist

{ fn

is_allowlisted (staking_pool_account_id :

AccountId)

->

bool ; } After creating the trait, we'll show two simple functions that will make a cross-contract call to an allowlist smart contract, asking if the accountmike.testnet is allowlisted. These methods will both returntrue using different approaches. First we'll look at the methods, then we'll look at the differences in callbacks. Note that for simplicity in this example, the values are hardcoded.

pub

const

XCC_GAS :

Gas

=

Gas (20_000_000_000_000) ; fn

get_allowlist_contract ( )

->

AccountId

{ "allowlist.demo.testnet" . parse ( ) . unwrap ( ) } fn

get_account_to_check ( )

->

AccountId

{ "mike.testnet" . parse ( ) . unwrap ( ) }

[near_bindgen]

impl

Contract

{ pub

fn

xcc_use_promise_result ( )

->

Promise

{ // Call the method is_allowlisted on the allowlisted contract. Static GAS is only attached to the callback. // Any unused GAS will be split between the function call and the callback since both have a default unused GAS weight of 1 // Attached deposit is defaulted to 0 for both the function call and the callback. ext_allowlist :: ext (get_allowlist_contract ( )) . is_allowlisted (get_account_to_check ( )) . then (Self :: ext (env :: current_account_id ( )) . with_static_gas (XCC_GAS) . callback_promise_result ( )) }

pub

fn

xcc_use_arg_macro (& mut

self)

->

Promise

{ // Call the method is_allowlisted on the allowlisted contract. Attach static GAS equal to XCC_GAS only for the callback. // Any unused GAS will be split between the function call and the callback since both have a default unused GAS weight of 1 // Attached deposit is defaulted to 0 for both the function call and the callback. ext_allowlist :: ext (get_allowlist_contract ( )) . is_allowlisted (get_account_to_check ( )) . then (Self :: ext (env :: current_account_id ( )) . with_static_gas (XCC_GAS) . callback_arg_macro ( )) } The syntax begins withext_allowlist::ext() showing that we're using the trait to call the method on the account passed intoext() . We then usewith_static_gas() to specify a base amount of GAS to attach to the call. We then call the methodis_allow_listed() and pass in the parameters we'd like to attach.

There are a couple things to note when doing these function calls:

1. You can attach a deposit of Ⓝ, in yoctoⓃ to the call by specifying the.with_attached_deposit()
2. method but it is defaulted to 0 (1 Ⓝ = 1000000000000000000000000 yoctoⓃ, or 1^24 yoctoⓃ).
3. You can attach a static amount of GAS by specifying the.with_static_gas()
4. method but it is defaulted to 0.
5. You can attach an unused GAS weight by specifying the.with_unused_gas_weight()
6. method but it is defaulted to 1. The unused GAS will be split amongst all the functions in the current execution depending on their weights. If there is only 1 function, any weight above 1 will result in all the unused GAS being attached to that function. If you specify a weight of 0, however, the unused GAS willnot
7. be attached to that function. If you have two functions, one with a weight of 3, and one with a weight of 1, the first function will get3/4
8. of the unused GAS and the other function will get1/4
9. of the unused GAS.

The two methods in the snippet above are very similar, except they will call separate callbacks in the smart contract,callback_promise_result andcallback_arg_macro . These two callbacks show how a value can be obtained.

[private]

pub

fn

callback_arg_macro (

[callback_unwrap]

val :

bool)

->

bool

{ val }

[private]

pub

fn

callback_promise_result ( )

->

bool

{ assert_eq! (env :: promise_results_count ( ) ,

1 ,

"ERR_TOO_MANY_RESULTS") ; match

env :: promise_result (0)

{ PromiseResult :: NotReady

=>

unreachable! ( ) , PromiseResult :: Successful (val)

=>

{ if

let

Ok (is_allowlisted)

=

near_sdk :: serde_json :: from_slice :: < bool

(& val)

{ is_allowlisted }

else

{ env :: panic_str ("ERR_WRONG_VAL_RECEIVED") } } , PromiseResult :: Failed

=>

env :: panic_str ("ERR_CALL_FAILED") , } } The first method uses a macro on the argument to cast the value into what's desired. In this approach, if the value is unable to be casted, it will panic. If you'd like to gracefully handle the error, you can either use the first approach, or use the#[callback_result] macro instead. An example of this can be seen below.

[private]

pub

fn

handle_callbacks (// New pattern, will gracefully handle failed callback results

[callback_result]

b :

Result < u8 ,

near_sdk :: PromiseError

,)

{ if b . is_err ( )

{ // ... } } The second method gets the value from the promise result and is essentially the expanded version of the# [callback_result] macro.

And that's it! Understanding how to make a cross-contract call and receive a result is an important part of developing smart contracts on NEAR. Two interesting references for using cross-contract calls can be found in thefungible token andnon-fungible token examples. Edit this page Last updatedonJan 24, 2024 byDamián Parrino Was this page helpful? Yes No

Previous Serialization Protocols Next Promises: Introduction