Cross-Contract Calls

Cross-contract calls allow your contract to interact with other deployed contracts. This is useful for:

1. Querying information from another contract.
2. Executing a method in another contract.

Cross-Contract Calls areIndependent

The method in which you make the call and the method in which you receive the result are different.

Cross-Contract Calls areAsynchronous

There is a delay between the call and the callback in which everyone can still interact with your contract.

Snippet: Querying Information

While making your contract, it is likely that you will want to query information from another contract. Below, you can see a basic example in which we query the greeting message from ourHello NEAR example.

- JavaScript
- Rust

contract-simple-ts/src/contract.ts loading ... See full example on GitHub * lib.rs * external.rs

contract-simple-rs/src/lib.rs loading ... See full example on GitHub contract-simple-rs/src/external.rs loading ... See full example on GitHub

Snippet: Sending Information

Calling another contract passing information is also a common scenario. Bellow you can see a method that interacts with theHello NEAR example to change its greeting message.

- JavaScript
- Rust

contract-simple-ts/src/contract.ts loading ... See full example on GitHub * lib.rs * external.rs

contract-simple-rs/src/lib.rs loading ... See full example on GitHub contract-simple-rs/src/external.rs loading ... See full example on GitHub

Promises

Cross-contract calls work by creating two promises in the network:

1. A promise to execute code in the external contract (Promise.create
2.).
3. A promise to call back adifferent
4. method in your contract with the result (Promise.then
5.).

Both promises take the same arguments:

- JavaScript
- Rust

NearPromise . new ("external_address") . functionCall ("method" ,

JSON . stringify (arguments) ,

DEPOSIT ,

GAS) ; external_trait :: ext ("external_address") . with_attached_deposit (DEPOSIT) . with_static_gas (GAS) . method (arguments) ; * The address of the contract you want to interact with * The method that you want to execute * The (encoded

*) arguments to pass to the method * The amount of GAS to use (deducted from theattached Gas *) * The amount of NEAR to attach (deducted fromyour contract's balance *)

tip Notice that the callback could be made toany contract. This means that, if you want, the result could be potentially handled by another contract. caution The fact that you are creating a Promise means that both the cross-contract call and callback willnot execute immediately . In fact:

- The cross-contract call will execute 1 or 2 blocks after your method finishescorrectly
- .
- The callback will then execute 1 or 2 blocks after theexternal
- method finishes (correctly or not
-)

Callback Method

If your method finishes correctly, then eventually your callback method will execute. This will happen whether the external contract finishessuccessfully or not . We repeat, if your original method finishes correctly, then your callback willalways execute .

In the callback method you will have access to the result, which contains two important arguments:

- status
- : Telling if the external method finished successfully or not
- buffer
- : Having the value returned by the external method (if any)

tip The callback methods in your contract must be public, so it can be called when the second promise executes. However, they should be only callable byyour contract . Always make sure to make it private by asserting that thepredecessor iscurrent_account_id . In rust this can be achieved using the#[private] decorator.

Checking Execution Status

- JavaScript
- Rust

contract-simple-ts/src/contract.ts loading ... See full example on GitHubcontract-simple-rs/src/lib.rs loading ... See full example on GitHub

Successful Execution

In case the call finishes successfully, the resulting object will have astatus of 1, and thebuffer will have the encoded result (if any). In order to recover the result you need to decode it from the resultingbuffer :

- JavaScript
- Rust

contract-simple-ts/src/contract.ts loading ... See full example on GitHubcontract-simple-rs/src/lib.rs loading ... See full example on GitHub

Failed Execution

If the external method fails (i.e. it panics), then your callback will beexecuted anyway . Here you need tomanually rollback any changes made in your contract during the original call. Particularly:

1. If the contract attached NEAR to the call, the funds are sent back tothe contract's account
2. .
3. If the original method made any state changes (i.e. changed or stored data), theywon't be automatically reverted
4. .

warning If your original method finishes correctly then the callback executeseven if the external method panics . Your state willnot rollback automatically, and NEAR willnot be returned to the signer automatically. Always make sure to check in the callback if the external method failed, and manually rollback any operation if necessary.

Security Concerns

While writing cross-contract calls there is a significant aspect to keep in mind: all the calls areindependent andasynchronous . In other words:

- The method in which you make the call and method for the callback areindependent

- .
- There is adelay between the call and the callback
- , in which people can still interact with the contract

This has important implications on how you should handle the callbacks. Particularly:

1. Make sure you don't leave the contract in a exploitable state between the call and the callback.
2. Manually rollback any changes to the state in the callback if the external call failed.

We have a wholesecurity section dedicated to these specific errors, so please go and check it.

warning Not following these basic security guidelines could expose your contract to exploits. Please check thesecurity section , and if still in doubtjoin us in Discord . Edit this page Last updatedonMar 12, 2024 bymatiasbenary Was this page helpful? Yes No

Previous Transfers & Actions Next ✅ Checklist