Transfers & Actions

Smart contracts can perform specificActions such as transferring NEAR, or calling other contracts.

An important property ofActions is that they can be batched together when acting on the same contract.Batched actions act as a unit: they execute in the samereceipt , and ifany fails , then theyall get reverted .

info Actions can be batched only when they act on thesame contract . You can batch calling two methods on a contract, butcannot call two methods on different contracts.

Transfer NEAR Ⓝ

You can send NEAR from your contract to any other account on the network. The Gas cost for transferring NEAR is fixed and is based on the protocol's genesis config. Currently, it costs~0.45 TGas .

- JavaScript
- Rust

import

{

NearBindgen ,

NearPromise , call }

from

'near-sdk-js' import

{

AccountId

}

from

'near-sdk-js/lib/types'

@ NearBindgen ({ }) class

Contract { @ call ({ }) transfer ({ to , amount } :

{

to :

AccountId ,

amount : bigint })

{ NearPromise . new (to) . transfer (amount) ; } } use

near_sdk :: borsh :: { self ,

BorshDeserialize ,

BorshSerialize } ; use

near_sdk :: { near_bindgen ,

AccountId ,

Promise ,

Balance } ;

[near_bindgen]

[derive(Default, BorshDeserialize, BorshSerialize)]

pub

struct

Contract

{

}

[near_bindgen]

impl

Contract

{ pub

fn

transfer (& self , to :

AccountId , amount :

Balance) { Promise :: new (to) . transfer (amount) ; } } tip The only case where a transfer will fail is if the receiver account doesnot exist. caution Remember that your balance is used to cover for the contract's storage. When sending money, make sure you always leave enough to cover for future storage needs.

Function Call

Your smart contract can call methods in another contract. In the snippet bellow we call a method in a deployed Hello NEAR contract, and check if everything went right in the callback.

- JavaScript
- Rust

import

{

NearBindgen , near , call , bytes ,

NearPromise

}

from

'near-sdk-js' import

{

AccountId

}

from

'near-sdk-js/lib/types'

const

HELLO_NEAR :

AccountId

=

"hello-nearverse.testnet" ; const

NO_DEPOSIT : bigint =

BigInt (0) ; const

CALL_GAS : bigint =

BigInt ("10000000000000") ;

@ NearBindgen ({ }) class

Contract

{ @ call ({ }) call_method ({ }) :

NearPromise

{ const args =

bytes (JSON . stringify ({

message :

"howdy"

}))

return

NearPromise . new (HELLO_NEAR) . functionCall ("set_greeting" , args ,

NO_DEPOSIT ,

CALL_GAS) . then (NearPromise . new (near . currentAccountId ( )) . functionCall ("callback" ,

bytes (JSON . stringify ({ })) ,

NO_DEPOSIT ,

CALL_GAS)) . asReturn ( ) }

@ call ({ privateFunction :

true }) callback ({ }) : boolean { let result , success ;

try { result = near . promiseResult (0) ; success =

true

} catch { result =

undefined ; success =

false

}

if

(success)

{ near . log (Success!) return

true }

else

{ near . log ("Promise failed...") return

false } } } use

near_sdk :: borsh :: { self ,

BorshDeserialize ,

BorshSerialize } ; use

near_sdk :: { near_bindgen , env , log ,

Promise ,

Gas ,

PromiseError } ; use

serde_json :: json ;

[near_bindgen]

[derive(Default, BorshDeserialize, BorshSerialize)]

pub

struct

Contract

{

}

const

HELLO_NEAR :

& str

=

"hello-nearverse.testnet" ; const

NO_DEPOSIT :

u128

=

0 ; const

CALL_GAS :

Gas

=

Gas (5_000_000_000_000) ;

[near_bindgen]

impl

Contract

{ pub

fn

call_method (& self) { let args =

json! ({

"message" :

"howdy" . to_string ( )

}) . to_string ( ) . into_bytes ( ) . to_vec ( ) ;

Promise :: new (HELLO_NEAR . parse ( ) . unwrap ( )) . function_call ("set_greeting" . to_string ( ) , args ,

NO_DEPOSIT ,

CALL_GAS) . then (Promise :: new (env :: current_account_id ( )) . function_call ("callback" . to_string ( ) ,

Vec :: new ( ) ,

NO_DEPOSIT ,

CALL_GAS)) ; }

pub

fn

callback (& self ,

[callback_result]

result :

Result < ( ) ,

PromiseError

>) { if result . is_err ( ) { log! ("Something went wrong") } else { log! ("Message changed") } } } warning The snippet showed above is a low level way of calling other methods. We recommend make calls to other contracts as explained in theCross-contract Calls section .

Create a Sub Account

Your contract can create direct sub accounts of itself, for example,user.near can createsub.user.near .

Accounts doNOT have control over their sub-accounts, since they have their own keys.

Sub-accounts are simply useful for organizing your accounts (e.g.dao.project.near ,token.project.near).

- JavaScript
- Rust

import

{

NearBindgen , near , call ,

NearPromise

}

from

'near-sdk-js'

const

MIN_STORAGE : bigint =

BigInt ("100000000000000000000")

// 0.001Ⓝ

@ NearBindgen ({ }) class

Contract

{ @ call ({ payableFunction : true }) create ({ prefix } : { prefix :

String })

{ const account_id =

{ prefix } . { near . currentAccountId ( ) }

NearPromise . new (account_id) . createAccount ( ) . transfer (MIN_STORAGE) } } use

near_sdk :: borsh :: { self ,

BorshDeserialize ,

BorshSerialize } ; use

near_sdk :: { near_bindgen , env ,

Promise ,

Balance } ;

[near_bindgen]

[derive(Default, BorshDeserialize, BorshSerialize)]

pub

struct

Contract

{

}

const

MIN_STORAGE :

Balance

=

1_000_000_000_000_000_000_000_000 ;

//0.001Ⓝ

[near_bindgen]

impl

Contract

{ pub

fn

create (& self , prefix :

String) { let account_id = prefix +

"."

+

& env :: current_account_id ( ) . to_string ( ) ; Promise :: new (account_id . parse ( ) . unwrap ( )) . create_account ( ) . transfer (MIN_STORAGE) ; } } tip Notice that in the snippet we are transferring some money to the new account for storage caution When you create an account from within a contract, it has no keys by default. If you don't explicitlyadd keys to it

ordeploy a contract on creation then it will belocked .

Creating Other Accounts

Accounts can only create immediate sub-accounts of themselves.

If your contract wants to create a.mainnet or.testnet account, then it needs tocall thecreate_account method ofnear ortestnet root contracts.

- JavaScript
- Rust

import

{

NearBindgen , near , call , bytes ,

NearPromise

}

from

'near-sdk-js'

const

MIN_STORAGE : bigint =

BigInt ("182000000000000000000") ;

//0.00182Ⓝ const

CALL_GAS : bigint =

BigInt ("28000000000000") ;

@ NearBindgen ({ }) class

Contract

{ @ call ({ }) create_account ({ account_id , public_key } : { account_id :

String ,

public_key :

String })

{ const args =

bytes (JSON . stringify ({ "new_account_id" : account_id , "new_public_key" : public_key }))

NearPromise . new ("testnet") . functionCall ("create_account" , args ,

MIN_STORAGE ,

CALL_GAS) ; } } use

near_sdk :: borsh :: { self ,

BorshDeserialize ,

BorshSerialize } ; use

near_sdk :: { near_bindgen ,

Promise ,

Gas ,

Balance

} ; use

serde_json :: json ;

[near_bindgen]

[derive(Default, BorshDeserialize, BorshSerialize)]

pub

struct

Contract

{

}

const

CALL_GAS :

Gas

=

Gas (28_000_000_000_000) ; const

MIN_STORAGE :

Balance

=

1_820_000_000_000_000_000_000 ;

//0.00182Ⓝ

[near_bindgen]

impl

Contract

{ pub

fn

create_account (& self , account_id :

String , public_key :

String) { let args =

json! ({ "new_account_id" : account_id , "new_public_key" : public_key , }) . to_string ( ) . into_bytes ( ) . to_vec ( ) ;

// Use "near" to create mainnet accounts Promise :: new ("testnet" . parse ( ) . unwrap ( )) . function_call ("create_account" . to_string ( ) , args ,

MIN_STORAGE ,

CALL_GAS) ; } }

Deploy a Contract

When creating an account you can also batch the action of deploying a contract to it. Note that for this, you will need to pre-load the byte-code you want to deploy in your contract.

- Rust

use

near_sdk :: borsh :: { self ,

BorshDeserialize ,

BorshSerialize } ; use

near_sdk :: { near_bindgen , env ,

Promise ,

Balance } ;

[near_bindgen]

[derive(Default, BorshDeserialize, BorshSerialize)]

pub

struct

Contract

{

}

const

MIN_STORAGE :

Balance

=

1_100_000_000_000_000_000_000_000 ;

//1.1Ⓝ const

HELLO_CODE :

& [u8]

=

include_bytes! ("./hello.wasm") ;

[near_bindgen]

impl

Contract

{ pub

fn

create_hello (& self , prefix :

String) { let account_id = prefix +

"."

+

& env :: current_account_id ( ) . to_string ( ) ; Promise :: new (account_id . parse ( ) . unwrap ( )) . create_account ( ) . transfer (MIN_STORAGE) . deploy_contract (HELLO_CODE . to_vec ( )) ; } } tip If an account with a contract deployed

doesnot have any access keys, this is known as a locked contract. When the account is locked, it cannot sign transactions therefore, actions canonly be performed fromwithin the contract code.

Add Keys

When you use actions to create a new account, the created account does not have anyaccess keys , meaning that itcannot sign transactions (e.g. to update its contract, delete itself, transfer money).

There are two options for adding keys to the account:

1. add_access_key
2. : adds a key that can only call specific methods on a specified contract.
3. add_full_access_key
4. : adds a key that has full access to the account.
5. JavaScript
6. Rust

import

{

NearBindgen , near , call ,

NearPromise

}

from

'near-sdk-js' import

{

PublicKey

}

from

'near-sdk-js/lib/types'

const

MIN_STORAGE : bigint =

BigInt ("100000000000000000000")

// 0.001Ⓝ

@ NearBindgen ({ }) class

Contract

{ @ call ({ }) create_hello ({ prefix , public_key } : { prefix :

String ,

public_key :

PublicKey })

{ const account_id =

{ prefix } . { near . currentAccountId ( ) }

NearPromise . new (account_id) . createAccount ( ) . transfer (MIN_STORAGE) . addFullAccessKey (public_key) } } use

near_sdk :: borsh :: { self ,

BorshDeserialize ,

BorshSerialize } ; use

near_sdk :: { near_bindgen , env ,

Promise ,

Balance ,

PublicKey } ;

[near_bindgen]

[derive(Default, BorshDeserialize, BorshSerialize)]

pub

struct

Contract

{

}

const

MIN_STORAGE :

Balance

=

1_100_000_000_000_000_000_000_000 ;

//1.1Ⓝ const

HELLO_CODE :

& [u8]

=

include_bytes! ("./hello.wasm") ;

[near_bindgen]

impl

Contract

{ pub

fn

create_hello (& self , prefix :

String , public_key :

PublicKey) { let account_id = prefix +

"."

+

& env :: current_account_id ( ) . to_string ( ) ; Promise :: new (account_id . parse ( ) . unwrap ( )) . create_account ( ) . transfer (MIN_STORAGE) . deploy_contract (HELLO_CODE . to_vec ( )) . add_full_access_key (public_key) ; } } Notice that what you actually add is a "public key". Whoever holds its private counterpart, i.e. the private-key, will be able to use the newly access key.

tip If an account with a contract deployed doesnot have any access keys, this is known as a locked contract. When the account is locked, it cannot sign transactions therefore, actions canonly be performed fromwithin the contract code.

Delete Account

There are two scenarios in which you can use thedelete_account action:

1. As thelast
2. action in a chain of batched actions.

3. To make your smart contract delete its own account.

4. JavaScript

5. Rust

import

{

NearBindgen , near , call ,

NearPromise

}

from

'near-sdk-js' import

{

AccountId

}

from

'near-sdk-js/lib/types'

const

MIN_STORAGE : bigint =

BigInt ("10000000000000000000")

// 0.001Ⓝ

@ NearBindgen ({ }) class

Contract

{ @ call ({ }) create_delete ({ prefix , beneficiary } : { prefix :

String ,

beneficiary :

AccountId })

{ const account_id =

{ prefix } . { near . currentAccountId ( ) }

NearPromise . new (account_id) . createAccount ( ) . transfer (MIN_STORAGE) . deleteAccount (beneficiary) }

@ call ({ }) self_delete ({ beneficiary } : { beneficiary :

AccountId })

{ NearPromise . new (near . currentAccountId ( )) . deleteAccount (beneficiary) } } use

near_sdk :: borsh :: { self ,

BorshDeserialize ,

BorshSerialize } ; use

near_sdk :: { near_bindgen , env ,

Promise ,

Balance ,

AccountId } ;

[near_bindgen]

[derive(Default, BorshDeserialize, BorshSerialize)]

pub

struct

Contract

{

}

const

MIN_STORAGE :

Balance

=

1_000_000_000_000_000_000_000 ;

//0.001Ⓝ

[near_bindgen]

impl

Contract

{ pub

fn

create_delete (& self , prefix :

String , beneficiary :

AccountId) { let account_id = prefix +

"."

+

& env :: current_account_id ( ) . to_string ( ) ; Promise :: new (account_id . parse ( ) . unwrap ( )) . create_account ( ) . transfer (MIN_STORAGE) . delete_account (beneficiary) ; }

pub

fn

self_delete (beneficiary :

AccountId) { Promise :: new (env :: current_account_id ( )) . delete_account (beneficiary) ; } } Token Loss If the beneficiary account does not exist the funds will bedispersed among validators . Token Loss Donot usedelete to try fund a

new account. Since the account doesn't exist the tokens will be lost. Edit this page Last updatedonJan 31, 2024 bygagdiez Was this page helpful? Yes No

Previous State & Data Structures Next Cross-Contract Calls