Notes on Serialization

Smart contracts need to be able tocommunicate complex data in a simple way, while alsoreading and storing such data into their states efficiently.

To achieve such simple communication and efficient storage, smart contracts morph the data from their complex representation into simpler ones.

This process of translatingcomplex objects into simpler single-value representations is calledserialization . NEAR uses two serialization formatsJSON andBorsh .

1. JSON
2. is used to serialize the contract's input/output during a function call
3. Borsh
4. is used to serialize the contract's state.

Overview of Serialization Formats

Lets give a quick overview of both serialization formats, including their pros and cons, as well as an example on how their serializations look like.

JSON

: Objects to Strings

Features

- Self-describing format
- Easy interoperability with JavaScript
- Multiple implementations readily available
- But... it is not efficient both in computational times and resulting size

Example

Example { number : i32 =

2 ; arr :

Vector < i32

=

[0 ,

1] ; }

// serializes to "{\"number\": 2, \"arr\": [0, 1]}"

Borsh

: Objects to Bytes

Features

- Compact, binary format built to be efficiently (de)serialized
- Strict and canonical binary representation
- Less overhead: it does not need to store attributes names
- But... it is necessary to know the schema to (de)serialize the data

Example

Example { number : i32 =

2 ; arr :

Vector < i32

=

[0 ,

1] ; }

// serializes into [2 ,

0 ,

0 ,

0 ,

2 ,

0 ,

0 ,

0 ,

0 ,

0 ,

0 ,

0 ,

1 ,

0 ,

0 ,

0]

Serializing Input & Output

NEAR contracts can implement methods that both take and return complex objects. In order to handle this data in a simple way, JSON serialization is used.

Using JSON makes it easier for everyone to talk with the contracts, since most languages readily implement a JSON (de)serializer.

Example

Let's look at this example, written only for educational purposes:

[derive(Serialize)]

[serde(crate =

"near_sdk::serde")] pub

struct

A

{ pub a_number :

i32 , pub b_number :

u128 }

[derive(Serialize)]

[serde(crate =

"near_sdk::serde")] pub

struct

B

{ pub success :

bool , pub other_number :

i32 }

pub

fn

method (& self , struct_a :

A) :

B

{ return

B { true ,

0 } }

Receiving Data

When a user calls themethod , the contract receives the arguments encoded as a JSON string (e.g."{\"a_number\":0, \"b_number\":\"100\"}"), and proceed to (de)serialize them into the correct object (A{0, 100}) .

Returning Data

When returning the result, the contract will automatically encode the objectB{true, 0} into its JSON serialized value:"{\"success\":true, \"other_number\":0}" and return this string.

JSON Limitations Since JSON is limited to52 bytes numbers, you cannot useu64 /u128 as input or output. JSON simply cannot serialize them. Instead, you must useStrings .

TheNEAR SDK RS currently implements thenear_sdk::json_types::{U64, I64, U128, I128} that you can use for input / output of data.

Borsh: State Serialization

Under the hood smart contracts store data using simplekey/value pairs . This means that the contract needs to translate complex states into simple key-value pairs.

For this, NEAR contracts use borsh which is optimized for (de)serializing complex objects into smaller streams of bytes.

SDK-JS still uses json The JavaScript SDK uses JSON to serialize objects in the state, but the borsh implementation should arrive soon

Example

Let's look at this example, written only for educational purposes:

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize, PanicOnDefault)]

pub

struct

Contract

{ string :

String , vector :

Vector < u8

}

[near_bindgen]

impl

Contract

{

[init]

pub

fn

init (string :

String , first_u8 :

u8)

->

Self

{ let

mut vector :

Vector < u8

=

Vector :: new ("prefix" . as_bytes ( )) ; vector . push (& first_u8) ;

Self

{ string , vector } }

pub

fn

change_state (& mut

self , string :

String , number :

u8)

{ self . string = string ; self . vector . push (& number) ; } }

Empty State On Deploy

If we deploy the contract into a new account and immediately ask for the state we will see it is empty:

near view-state CONTRACT --finality optimistic

Result is: []

Initializing the State

If we initialize the state we can see how Borsh is used to serialize the state

initialize with the string "hi" and 0

near call CONTRACT init '{"string":"hi", "first_u8":0}' --accountId CONTRACT

check the state

near view-state CONTRACT --utf8 --finality optimistic

Result is:

[

{

key: 'STATE',

value: '\x02\x00\x00\x00hi\x01\x00\x00\x00\x00\x00\x00\x00\x06\x00\x00\x00prefix'

},

{ key: 'prefix\x00\x00\x00\x00\x00\x00\x00\x00', value: '\x00' }

]

The first key-value is:

key :

'STATE' value :

'\x02\x00\x00\x00hi\x01\x00\x00\x00\x00\x00\x00\x00\x06\x00\x00\x00prefix' Since theContract has a structurestring, Vector the value is interpreted as:

[2, 0, 0, 0, "h", "i"] -> Thestring has 2 elements: "h" and "i". [1, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, "prefix"] -> The Vector has 1 element, and to see the values search for keys that start with (the 6 bytes prefix): "prefix" Then, the second key-value shows the entries of theVector denoted by the"prefix" string:

key :

'prefix\x00\x00\x00\x00\x00\x00\x00\x00' value :

'\x00'

Modifying the State

If we modify the stored string and add a new number, the state changes accordingly:

near call CONTRACT change_state '{"string":"bye", "number":1}' --accountId CONTRACT

Result is

[

{

key: 'STATE',

value: '\x03\x00\x00\x00bye\x02\x00\x00\x00\x00\x00\x00\x00\x06\x00\x00\x00prefix'

},

{ key: 'prefix\x00\x00\x00\x00\x00\x00\x00\x00', value: '\x00' },

{ key: 'prefix\x01\x00\x00\x00\x00\x00\x00\x00', value: '\x01' }

]

We can see that theSTATE key changes to reflect the storage of the new string (bye), and that the vector now has 2 elements.

At the same time, a new key-value was added adding the new vector entry: the1u8 we just added.

Deserialization Error

When somebody invokes a smart contract method, the first step for the contract is to deserialize its own state.

In the example used above, the contract will start by reading theSTATE key and try to deserialize its value into an objectContract{string: String, vector: Vector}.

If you deploy a contract into the account with a different Contract structure, then the contract will fail to deserialize theSTATE key and panicCannot deserialize the contract state .

To solve this, you can either:

1. Rollback to the previous contract code
2. Implement a method tomigrate the contract's state Edit this page Last updatedonApr 10, 2024 bygagdiez Was this page helpful? Yes No Need some help?Chat with us or check ourDev Resources ! Twitter Telegram Discord Zulip

Previous ✅ Checklist Next Introduction