Serialization Protocols

Serialization formats within the SDK define how data structures are translated into bytes which are needed for passing data into methods of the smart contract or storing data in state. For the case of method parameters,JSON (default) andBorsh are supported with the SDK and for storing data on-chain Borsh is used.

The qualities of JSON and Borsh are as follows:

JSON:

- Human-readable
- Self-describing format (don't need to know the underlying type)
- Easy interop with JavaScript
- Less efficient size and (de)serialization

Borsh:

- Compact, binary format that's efficient for serialized data size
- Need to know data format or have a schema to deserialize data
- Strict and canonical binary representation
- Fast and less overhead in most cases

In general, JSON will be used for contract calls and cross-contract calls for a better DevX, where Borsh can be used to optimize using less gas by having smaller parameter serialization and less deserialization computation within the contract.

Overriding Serialization Protocol Default

The result and parameter serialization can be opted into separately, but all parameters must be of the same format (can't serialize some parameters as borsh and others as JSON). An example of switching both the result and parameters to borsh is as follows:

[result_serializer(borsh)]

pub

fn

sum_borsh (

[serializer(borsh)]

a :

u32 ,

[serializer(borsh)]

b :

u32)

->

u32

{ a + b } Where theresult_serializer(borsh) annotation will override the default result serialization protocol from JSON to borsh and theserializer(borsh) annotations will override the parameter serialization.

Example

A simple demonstration of getting aBorsh-serialized , base64-encoded value from a unit test:

src/lib.rs loading ... See full example on GitHub The following snippet shows a simple function that takes this value from a frontend or CLI. Note: this method doesn't have a return value, so the#[result_serializer(borsh)] isn't needed.

src/lib.rs loading ... See full example on GitHub Note that this is using this simple struct:

src/lib.rs loading ... See full example on GitHub To call this with NEAR CLI, use a command similar to this:

- near-cli
- near-cli-rs

near call rust-status-message.demo.testnet set_status_borsh --base64 'DAAAAEFsb2hhIGhvbnVhIQ==' --accountId demo.testnet near contract call-function as-transaction rust-status-message.demo.testnet set_status_borsh base64-args 'DAAAAEFsb2hhIGhvbnVhIQ==' prepaid-gas '30 TeraGas' attached-deposit '0 NEAR' sign-as demo.testnet network-config testnet sign-with-keychain send See more details inthis GitHub gist fromMarcelo .

JSON wrapper types

To help with serializing certain types to JSON which have unexpected or inefficient default formats, there are some wrapper types innear_sdk::json_types that can be used.

Because JavaScript only supports integers to value2^53 - 1 , you will lose precision if deserializing the JSON integer is above this range. To counteract this, you can use theI64 ,U64 ,I128 , andU128 in place of the native types for these parameters or result to serialize the value as a string. By default, all integer types will serialize as an integer in JSON.

You can convert fromU64 tou64 and back usingstd::convert::Into , e.g.

[near_bindgen]

impl

Contract

{ pub

fn

mult (& self , a :

U64 , b :

U64)

->

U128

{ let a :

u64

= a . into ( ) ; let b :

u64

= b . into ( ) ; let product =

u128 :: from (a)

*

u128 :: from (b) ; product . into ( ) } } You can also access inner values and using.0 :

[near_bindgen]

impl Contract { pub fn mult(&self, a: U64, b: U64) -> U128 { - let a: u64 = a.into(); + let a = a.0; - let b: u64 = b.into(); + let b = b.0; let product = u128::from(a) * u128::from(b); product.into() } } And you can cast the lower-caseu variants to upper-caseU variants usingU64(...) andU128(...) :

[near_bindgen]

impl Contract { pub fn mult(&self, a: U64, b: U64) -> U128 { let a = a.0; let b = b.0; let product = u128::from(a) * u128::from(b); - product.into() + U128(product) } } Combining it all:

[near_bindgen]

impl

Contract

{ pub

fn

mult (& self , a :

U64 , b :

U64)

->

U128

{ U128 (u128 :: from (a .0)

*

u128 :: from (b .0)) } } Although there are these JSON wrapper types included with the SDK, any custom type can be used, as long as it implementsserde serialize and deserialize respectively. All of these types just override the JSON format and will have a consistentborsh serialization and deserialization as the inner types.

Base64VecU8

Another example of a type you may want to override the default serialization of isVecwhich represents bytes in Rust. By default, this will serialize as an array of integers, which is not compact and very hard to use. There is a wrapper typeBase64VecU8 which serializes and deserializes to aBase-64 string for more compact JSON serialization.

Example here:

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize, PanicOnDefault)]

pub

struct

Contract

{ // Notice, internally we store Vec<u8> pub data :

Vec < u8

, }

[near_bindgen]

impl

Contract

{

[init]

pub

fn

new (data :

Base64VecU8)

->

Self

{ Self

{ data : data . into ( ) , } } pub

fn

get_data (self)

->

Base64VecU8

{ self . data . into ( ) } }Edit this page Last updatedonJan 19, 2024 byDamián Parrino Was this page helpful? Yes No