CosmWasm (general)

Overview

CosmWasm is a smart contract platform focusing on security, performance, and interoperability It is the only smart contracting platform for public blockchains with heavy adoption outside of the EVM world.

Key features of CosmWasm are:

Secure

CosmWasm architecture prevents almost all the known risk vectors of Ethereum.

Powerful

CosmWasm runs the Web Assembly, Wasm virtual machine guarantees high performance.

Interoprable

CosmWasm was built for multi-chain, cross-chain world, deeply integrated with IBC (Inter-blockchain communication).

Smart contract language

CosmWasm smart contracts are written inRust(opens in a new tab) programming language. Here's a good reference if you would like to make adeep dive(opens in a new tab) .

Rust

Why Rust?

Performance. Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

Reliability. Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.

Productivity. Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

Example CosmWasm smart contract

// cosmwasm_std is a standard library for smart contracts. // It provides essential utilities for communication with the outside world // and a couple of helper functions and types. // Every smart contract uses this dependency. use cosmwasm_std :: { entry_point, to_binary, Binary , Deps , DepsMut , Empty , Env , MessageInfo , Response , StdResult , }; // serde is a serialization/deserilization library use serde :: { Deserialize , Serialize };

// Query response data structure

[derive(

Serialize , Deserialize)] struct

QueryResp { message :

String , }

// Typical Rust application starts with the fn main() function called by // the operating system. Smart contracts are not significantly different. // When the message is sent to the contract, a function called "entry point" // is called. Unlike native applications, which have only a single main entry // point, smart contracts have a couple corresponding to different message types: // instantiate, execute, query, sudo, migrate and more

[entry_point]

// instantiate is called once per smart contract lifetime - you can think about // it as a constructor or initializer of a contract. pub

fn

instantiate (// DepsMut is a utility type for communicating with the outer world - // it allows querying and updating the contract state, // querying other contracts state, and gives access to an Api object with // a couple of helper functions for dealing with CW addresses. _deps :

DepsMut , // Env is an object representing the blockchains state when executing // the message - the chain height and id, current timestamp, and the called // contract address. _env :

Env , // MessageInfo contains metainformation about the message which triggered // an execution - an address that sends the message, and chain native // tokens sent with the message. _info :

MessageInfo , // Empty is the message triggering execution itself, it is Empty type // that represents {} JSON, but the type of this argument can be anything // that is deserializable. _msg :

Empty ,) ->

StdResult < Response

{ Ok (Response :: new ()) }

// Another entry point

[entry_point]

pub

fn

query (// Deps object is readonly as opposed to DevMut above. // That is because the query can never alter the smart contract's internal // state. It can only read the state. It comes with some consequences - // for example, it is impossible to implement caching for future queries // (as it would require some data cache to write to). _deps :

Deps , _env :

Env , _msg :

Empty) ->

StdResult < Binary

{ let resp =

QueryResp { message :

"Hello World" . to_owned (), };

to_binary (& resp) }

Deploying a smart contract on Sei

Let’s create a simple smart contract project from template.

Assuming you have a recent version of Rust and Cargo installed (viarustup(opens in a new tab)), then the following should get you a new repo to start a contract:

Installcargo-generate(opens in a new tab) and cargo-run-script.

cargo

install

cargo-generate

--features

vendored-openssl cargo

install

cargo-run-script Now, let’s use it to create your new contract project. Go to the folder in which you want to place it and run:

cargo

generate

--git

https://github.com/CosmWasm/cw-template.git

--name

counter Choosefalse when askedWould you like to generate the minimal template? .

This should create acounter contract project with source code generated inside folder with the same name.

Once inside the project openCargo.toml and remove features we won’t need.

cosmwasm-std

=

{

version

=

"2.0.1" ,

features

= ["cosmwasm_1_3" ,

Enable this if you only deploy to chains that have CosmWasm 1.4 or higher

"cosmwasm_1_4",

] } modify this line to

cosmwasm-std

=

"2.0.0" Before proceeding further let’s test the contract to make sure code is intact. Run

cargo

test You should see output similar to

Finished

test [unoptimized +

debuginfo]

target (s) in

0.09 s Running

unittests

src/lib.rs (target/debug/deps/c3-777d376b6d32a663)

running

4

tests test

contract::tests::proper_initialization

...

ok test

contract::tests::increment

...

ok test

contract::tests::reset

...

ok test

integration_tests::tests::count::count

...

ok

test

result:

ok.

4

passed ; 0

failed ; 0

ignored ; 0

measured ; 0

filtered

out ; finished

in

0.00 s

Running

unittests

src/bin/schema.rs (target/debug/deps/schema-283e665754e86143)

running

0

tests

test

result:

ok.

0

passed ; 0

failed ; 0

ignored ; 0

measured ; 0

filtered

out ; finished

in

0.00 s

Doc-tests

c3

running

0

tests

test

result:

ok.

0

passed ; 0

failed ; 0

ignored ; 0

measured ; 0

filtered

out ; finished

in

0.00 s Now, important steps involved in the contract deployment are building and optimization. Optimization is important to make sure contract will take least amount of space on blockchain possible and will require less gas.Optimizer(opens in a new tab) project has been created specifically for that purpose.

Our generated project contains already an alias command that we could run to invoke building and optimization:

Catgo.toml

[package.metadata.scripts] optimize =

"""docker run --rm -v "(pwd)":/code \ --mount type=volume,source="(basename "(pwd)")_cache",target=/target \ --mount type=volume,source=registry_cache,target=/usr/local/cargo/registry \ cosmwasm/optimizer:0.15.0 """ So we just run:

cargo

run-script

optimize After command runs successfully, we could fund our wasm artifact inartifact directory.

We could also verify it by running acosmwasm-check command:

cosmwasm-check

artifacts/counter.wasm

Available

capabilities:

{ "stargate" ,

"cosmwasm_1_1" ,

"cosmwasm_1_3" ,

"cosmwasm_2_0" ,

"cosmwasm_1_4" ,

"iterator" ,

"cosmwasm_1_2" ,

"staking" }

artifacts/counter.wasm:

pass

All

contracts (1) passed checks ! Now we are ready to deploy our contract.

Let’s deploy our contract to Sei test networkatlantic-2 . Should you choose another Sei network, it can be found in our registryhere(opens in a new tab) .

seid

tx

wasm

store

artifacts/counter.wasm

--from SEI_WALLET_ADDRESS --node

https://rpc-testnet.sei-apis.com

--chain-id

atlantic-2

-b

block

--fees=200000usei

--gas=2000000 Note the code in events:

-

events:

attributes:

key:

action value:

/cosmwasm.wasm.v1.MsgStoreCode

key:

module value:

wasm

key:

sender value: type : message -

attributes:

key:

code_id value:

"8363"

<

Code

ID type : store_code log:

"" msg_index:

0 Now lets, instantiate the contract:

seid

tx

wasm

instantiate

8363

'{"count":1}'

-y

--no-admin

--label

counter

--from SEI_WALLET_ADDRESS --node

https://rpc-testnet.sei-apis.com

--chain-id

atlantic-2

-b

block

--fees=40000usei

--gas=2000000 Note the contract address in the output.

Query the contract:

seid

q

wasm

contract-state

smart CONTRACT_ADDRESS '{"get_count": {}}'

--node

https://rpc-testnet.sei-apis.com Response should look like:

data: count:

1 Now lets callincrement function:

seid

tx

wasm

execute CONTRACT_ADDRESS '{"increment": {}}'

--from SEI_WALLET_ADDRESS --node

https://rpc-testnet.sei-apis.com

--chain-id

atlantic-2

-b

block

--fees=4000usei If successful, we can now re-query contract state and see counter incremented:

data: count:

2

Calling contract from JS client

To call the contract from frontend in EVM environment you could useethers and@sei-js library:

import {WASM_PRECOMPILE_ABI , WASM_PRECOMPILE_ADDRESS} from

"@sei-js/evm" ;

const

signer

=

await

getEthSigner ();

if (! signer) { console .log ('No signer found'); return ; } const

contract

=

new

ethers .Contract (WASM_PRECOMPILE_ADDRESS ,

WASM_PRECOMPILE_ABI , signer);

const

counterContractAddress

=

CONTRACT_ADDRESS ;

const

queryJSON

= {get_count : {}} try { const

response

=

await

contract .query (counterContractAddress ,

toUtf8Bytes (JSON .stringify (queryJSON))); console .log (toUtf8String (response)); } catch (e) { console .log (e); } Last updated onMay 24, 2024 Building a frontend EVM (General)