Deps/DepsMut

An explainer of the dependecies inside of the CosmWasm code framework Deps (or the mutable versionDepsMut) holds all external dependencies of the contract, and is designed to allow easy dependency injection at runtime. Those external dependencies are as follows:

Storage

As the name suggests, storage allows reading and writing data in a contract's memory. Secret Network contracts use a key-value store for reading and writing data and provide three easy-to-use methods to do so:

- get(&self, key: &[u8]) -> Option>
- reads data stored in the specified key
- set(&self, key: &[u8], value: &[u8])
- which writes data to the storage in the specified key
- remove(&self, key: &[u8])
- which deletes a key alongside its data.
-

You may have noticed that storage uses byte arrays exclusively to read and write data, so to make it more user-friendly there are certain libraries that the community has developed. You can learn more about thishere .

Often the question of, "How much data can be stored in a contract?" is asked. The answer is that it's complicated. Technically there's no upper-bound on how much data a contract can store, but there are practical per-transaction limits.

Reading one byte of data "uses" 3 gas and writing one byte of data "uses" 30 gas, so with the current limit of 6_000_000 gas per block, the practical limit is that a contract can read — at most — 13.33Mb of data per transaction and can write — at most — 1.3Mb of data per transaction. This assumes that the contract doesn't do anything but read and write data, the more complex the contract, the lower the amount of data you can read and write. Note that the limits are per transaction, you could write several gigabytes of data if you wanted to but it would take several transactions.

Api

Api consists of a collection of callbacks to system functions defined outside of the wasm modules. Those are functions that are defined in the chain's code, this allows calling useful functions without needing to include them in the contract's code and assures a smaller binary size as well as a higher certainty that they're going to be implemented correctly.

- canonical_address(&self, human: &HumanAddr) -> StdResult
- Takes a human readable address and returns a binary representation of it.
- human_address(&self, canonical: &CanonicalAddr) -> StdResult
- Takes a canonical address and returns a human readble address. This is the inverse ofcanonical_address
- secp256k1_verify(&self, message_hash: &[u8], signature: &[u8], public_key: &[u8]) -> Result
- Verifies message hashes (hashed unsing SHA-256) against a signature, with the public key of the signer, using the secp256k1 elliptic curve digital signature parametrization / algorithm.
- secp256k1_recover_pubkey(&self, message_hash: &[u8], signature: &[u8], recovery_param: u8) -> Result
- Recovers a public key from a message hash and a signature.
- ed25519_verify(&self, message: &[u8], signature: &[u8], public_key: &[u8]) -> Result
- Verifies messages against a signature, with the public key of the signer, using the ed25519 elliptic curve digital signature parametrization / algorithm.
- ed25519_batch_verify(&self, messages: &[&[u8]], signatures: &[&[u8]], public_keys: &[&[u8]]) -> Result
- Performs batch Ed25519 signature verification.
- Batch verification asks whether all signatures in some set are valid, rather than asking whether each of them is valid. This allows sharing computations among all signature verifications, performing less work overall, at the cost of higher latency (the entire batch must complete), complexity of caller code (which must assemble a batch of signatures across work-items), and loss of the ability to easily pinpoint failing signatures.
- This batch verification implementation is adaptive, in the sense that it detects multiple signatures created with the same verification key, and automatically coalesces terms in the final verification equation.
- secp256k1_sign(&self, message: &[u8], private_key: &[u8]) -> Result
- Signs a message with a private key using the secp256k1 elliptic curve digital signature parametrization / algorithm.
- ed25519_sign(&self, message: &[u8], private_key: &[u8]) -> Result
- Signs a message with a private key using the ed25519 elliptic curve digital signature parametrisation / algorithm.
- (CosmWasm 1.0)
- debug
-
    - a function that allows for debug prints, which can be used during development to print to STDOUT in a testnet or LocalSecret environment (replacesdebug_print
- from CosmWasm v0.10)
- (New in Secret-Cosmwasm v1.1.10)
- gas_evaporate(evaporate: u32) -> u32

-
    - this API function allows a contract to consume a set amount of gas. Use together withcheck_gas
- in order to create contract calls that consume an exact amount of gas regardless of the code path taken.Documentation is here.
- (New in Secret-Cosmwasm v1.1.10)
- check_gas() -> u64
-
    - this API returns the current amount of gas consumed by a contract call.
-

Last updated7 months ago On this page *Storage * Api

Was this helpful? Edit on GitHub Export as PDF