Storage

An explainer on the varying storage frameworks for Secret contracts

How Storage Works

CosmWasm uses a key-value storage design. Smart contracts can store data in binary, access it through a storage key, edit it, and save it. Similar to a HashMap, each storage key is associated with a specific piece of data stored in binary. The storage keys are formatted as references to byte arrays (&[u8]).

One advantage of the key-value design is that a particular data value is only loaded when the user explicitly loads it using its storage key. This prevents any unnecessary data from being processed, saving resources.

Any type of data may be stored this way as long as the user can serialize/deserialize (serde) the data to/from binary. Doing this manually every single time is cumbersome and repetitive, this is why we have wrapper functions that does this serde process for us.

All the data is actually stored indeps.storage , and the examples below show how to save/load data to/from there with a storage key.

Storage Keys

Creating a storage key is simple, any way of generating a constant&[u8] suffices. People often prefer generating these keys from strings as shown in the example below.

```

Copy pubconstCONFIG_KEY:&[u8]=b"config";

```

For example, the above key is likely used to store some data related to core configuration values of the contract. The convention is that storage keys are often all created instate.rs , and then imported tocontract.rs . However, since storage keys are just constants, they could be declared anywhere in the contract.

The example above also highlights that storage keys are not meant to be secret nor hard to guess. Anyone who has the open source code can see what the storage keys are (and of course this is not enough for a user to load any data from the smart contract).

Storage Wrapper Functions

As mentioned above, serializing/deserializing data while loading/saving it with a key is cumbersome. This is why we often use wrapper functions written by community members. There are three common wrapper functions that are included in thestate.rs of most secret contract templates.

Saving To Storage

A commonly used wrapper function to save data is the following. This function overwrites previously saved data associated to that storage key.

```

Copy /// Returns StdResult<()> resulting from saving an item to storage /// /// # Arguments /// /// * storage - a mutable reference to the storage this item should go to /// * key - a byte slice representing the key to access the stored item /// * value - a reference to the item to store pubfnsave(storage:&mutS, key:&[u8], value:&T)->StdResult<()> { storage.set(key,&Bincode2::serialize(value)?); Ok(()) }

```

Note thatvalue can be of any Struct type. The only condition is that this Struct must derive the Serialize and Deserialize traits from serde with the following line above its Struct declaration.

```

Copy

# [derive(Serialize,Deserialize)

```

There are some structs that cannot be serialized/deserialized by the bincode2 struct these wrapper functions use! See JSON Storage Wrapper Functions section below to see what happens in this case.

Example

The wrapper may be used to save data in the following manner:

```

Copy letconfig=Config{ owner:deps.api.canonical_address(&env.message.sender)?, };

// Save data to storage save(&mutdeps.storage, CONFIG_KEY,&config)?;

```

Loading From Storage

A commonly used wrapper function to load data from storage is the following:

```

Copy /// Returns StdResult from retrieving the item with the specified key. Returns a /// StdError::NotFound if there is no item with that key /// /// #
```

Arguments /// /// * storage - a reference to the storage this item is in /// * key - a byte slice representing the key that accesses the stored item pubfnload(storage:&S, key:&[u8])->StdResult { Bincode2::deserialize(&storage .get(key) .ok_or_else(||StdError::not_found(type_name:())))?,) }

```

Note that this function throws an error if there is no data previously saved with that storage key.

Example

When loading data, rust must be told what Struct to expect after deserializing.

```

Copy letconfig:Config=load(&deps.storage, CONFIG_KEY)?;

```

Loading With may_load

In some instances you may be unsure whether there is any data stored with a particular key. In this case you want to usemay_load() which wraps any data inside within an option. ReturningNone if there is no value saved with that key, and returningSome(value) if there is some value saved. An example function for this is:

```

Copy /// Returns StdResult> from retrieving the item with the specified key. /// Returns Ok(None) if there is no item with that key /// /// # Arguments /// /// * storage - a reference to the storage this item is in /// * key - a byte slice representing the key that accesses the stored item pubfnmay_load(storage:&S, key:&[u8],)->StdResult> { matchstorage.get(key) { Some(value)=>Bincode2::deserialize(&value).map(Some), None=>Ok(None), } }

```

Example

One of the most common use cases of this function is when retrieving viewing keys for users. However, viewing keys use PrefixedStorage, and we will see this in the next section. So instead, I will show a line of code that retrieves a list of minters for an NFT.

```

Copy letminters:Vec= may_load(&deps.storage, MINTERS_KEY)?.unwrap_or_else(Vec::new);

```

This line of code returns the list of minters if it is saved to MINTERS_KEY, otherwise, it returns an empty list.

Removing From Storage

A commonly used wrapper function to remove saved data from storage is the following. This might be the only wrapper function that does not make anything more convenient, because there is no serialize/deserialize implemented.

```

Copy /// Removes an item from storage /// /// # Arguments /// /// * storage - a mutable reference to the storage this item is in /// * key - a byte slice representing the key that accesses the stored item pubfnremove(storage:&mutS, key:&[u8]) { storage.remove(key); }

```

Example

The following code removes minters. This code does not let you know if there was any previously saved data to that storage key.

```

Copy remove(&mutdeps.storage, MINTERS_KEY);

```

JSON Storage Wrapper Functions

The wrapper functions we learned above use bincode2 struct (fromsecret-toolkit ) to serde the data being saved/read on the smart contract. However, bincode2 uses floats when deserializing rust enum variants, thus, bincode2 cannot serde enum variants in cosmwasm. This is why cosmwasm uses Json serde by default, not bincode2.

The following is an example, from thereference SNIP-721 implementation , of a struct that cannot be saved/loaded by the wrapper functions we saw above because it uses an enum.

```

Copy /// permission to view token info/transfer tokens

[derive(Serialize,Deserialize,Clone,PartialEq,Debug)]

pubstructPermission{ /// permitted address pubaddress:CanonicalAddr, /// list of permission expirations for this address pubexpirations:[Option;3], }

/// at the given point in time and after, Expiration will be considered expired

[derive(Serialize,Deserialize,Clone,Copy,PartialEq,JsonSchema,Debug)]

[serde(rename_all="snake_case")]

pubenumExpiration{ /// expires at this block height AtHeight(u64), /// expires at the time in seconds since 01/01/1970 AtTime(u64), /// never expires Never, }

```

In these cases, we can use the Json struct fromsecret-toolkit to serde structs that use enums. This also creates the need for for new wrapper functions

Saving To Storage

```

Copy /// Returns StdResult<()> resulting from saving an item to storage using Json (de)serialization /// because bincode2 annoyingly uses a float op when deserializing an enum /// /// # Arguments /// /// * `storage` - a mutable reference to the storage this item should go to /// * `key` - a byte slice representing the key to access the stored item /// * `value` - a reference to the item to store pubfnjson_save(storage:&mutS, key:&[u8], value:&T,)->StdResult<()> { storage.set(key,&Json::serialize(value)?); Ok(()) }

```

The usage of this function is extremely similar to thesave wrapper function we discussed above .

bincode2 serde is more efficient than json serde

Loading from Storage

```

Copy /// Returns StdResult from retrieving the item with the specified key using Json /// (de)serialization because bincode2 annoyingly uses a float op when deserializing an enum. /// Returns a StdError::NotFound if there is no item with that key /// /// # Arguments /// /// * `storage` - a reference to the storage this item is in /// * `key` - a byte slice representing the key that accesses the stored item pubfnjson_load(storage:&S, key:&[u8])->StdResult { Json::deserialize(&storage .get(key) .ok_or_else(||StdError::not_found(type_name::()))?,) }

```

The usage of this function is extremely similar to theload wrapper function we discussed above .

Loading with may_load

```

Copy /// Returns StdResult> from retrieving the item with the specified key using Json /// (de)serialization because bincode2 annoyingly uses a float op when deserializing an enum. /// Returns Ok(None) if there is no item with that key /// /// # Arguments /// /// * `storage` - a reference to the storage this item is in /// * `key` - a byte slice representing the key that accesses the stored item pubfnjson_may_load(storage:&S, key:&[u8],)->StdResult> { matchstorage.get(key) { Some(value)=>Json::deserialize(&value).map(Some), None=>Ok(None), } }

```

The usage of this function is extremely similar to themay_load wrapper function we discussed above .

Theremove wrapper functionabove works the same because it doesn't serde

Last updated7 months ago On this page *How Storage Works * Storage Keys * Storage Wrapper Functions * Saving To Storage * Loading From Storage * Loading With may_load * Removing From Storage * JSON Storage Wrapper Functions * Saving To Storage * Loading from Storage * Loading with may_load

Was this helpful? Edit on GitHub Export as PDF
```