working-with-a-singleton-in-cosmwasm)

Singleton

Working with a Singleton in CosmWasm

Working with a Singleton in CosmWasm

In CosmWasm smart contracts, aSingleton is a specialized structure that combines the concepts ofPrefixedStorage andTypedStorage , allowing you to store and manage a single data entity efficiently under a specific storage key. This is particularly useful when you need to store just one instance of a state (such as contract configuration or global data) without the need for complex key management.

How Singleton Works:

```

Copy pubstructSingleton<'a,T> where T:Serialize+DeserializeOwned, { storage:&'amutdynStorage, key:Vec, data:PhantomData, }

```

- storage: &'a mut dyn Storage
- : TheSingleton
- holds a mutable reference to the storage layer, which is where the contract data is stored on-chain. This reference allows it to interact with and modify the stored data.
- key: Vec
- : TheSingleton
- operates on a single storage key, provided during its instantiation. This key is typically transformed usingto_length_prefixed
- to avoid collisions with other stored data, ensuring that the contract can safely manage this unique piece of state.
- data: PhantomData
- : Rust'sPhantomData
- is a marker for the data typeT
- without actually holding an instance of it. This allows theSingleton
- to enforce that it works with a specific type (T
-) that must implementSerialize
- andDeserializeOwned
- traits, which are essential for encoding/decoding the data stored in the singleton.

Key Features of Singleton:

1. No Need for Multiple Keys
2. : ASingleton
3. simplifies storage management by removing the need to manually manage multiple keys. The storage key is predefined in the constructor, and all operations (read, write, update) are tied to this single key. This ensures you are only ever working with one data object in storage.
4. Type-Safe Storage
5. : TheSingleton
6. usesTypedStorage
7. functionality, which ensures that the data stored and retrieved from the storage is type-safe. This means you always work with the exact type you've defined (in this case,T
8.), reducing the risk of errors during serialization and deserialization.
9. Avoids Key Collisions
10. : By applying theto_length_prefixed
11. transformation to the given storage key, theSingleton
12. ensures there are no name collisions in storage. This is particularly useful in larger contracts where multiple data points might otherwise share similar key names.

There are common wrapper functions that are included in thestate.rs of most secret contract templates, which aresave, load,

may_load, remove, andupdate .

save

Thesave function overwrites previously saved data associated to that storage key.save will serialize the model and store, returns an error on serialization issues:

```

Copy pubfnsave(&mutself, data:&T)->StdResult<()> { self.storage.set(&self.key,&to_vec(data)?); Ok(()) }

```

- Purpose
- : It takes a reference to the storage and the data to be saved (usually the contract's state).
- Usage
- : You callsave
- when you want to persist new data or update existing data in storage.

load

This function reads or retrieves data from the contract's state. It is commonly used when you need to access a piece of data stored in the contract, such as the current state or configuration.

```

Copy /// load will return an error if no data is set at the given key, or on parse error pubfnload(&self)->StdResult{ letvalue=self.storage.get(&self.key); must_deserialize(&value) }

```

- Purpose
- : It accesses the storage and loads the state associated with the provided key.
- Usage
- :load
- is used whenever the contract needs to read existing data, such as fetching the configuration or state before making any modifications.

may_load

```

Copy /// may_load will parse the data stored at the key if present, returns Ok(None) if no data there. /// returns an error on issues parsing pubfnmay_load(&self)->StdResult> { letvalue=self.storage.get(&self.key); may_deserialize(&value) }

```

- Purpose
- : This function attempts to retrieve and deserialize data from storage, returningOk(None)
- if no data exists at the given key. It returns an error only if deserialization fails.
- Usage
- :may_load
- is used when the existence of the data is optional. It allows you to safely check if data exists and handle cases where the data might not be present.
- Behavior
- : It fetches the data associated with the key and attempts to deserialize it usingmay_deserialize
- . If the key does not have any data, it returnsOk(None)
- instead of an error, making it more forgiving thanload
- .

remove

```

Copy pubfnremove(&mutself) { self.storage.remove(&self.key) }

```

- Purpose
- : This function deletes or removes data associated with a specific key from the contract’s storage.
- Usage

- :remove
- is used when you no longer need to store certain data in the contract and want to free up space by removing the key-value pair from storage.
- Behavior
- : It accesses thestorage
- object and deletes the data associated with the provided key.

update

This function loads, modifies, and then saves data back to the contract's state. It’s commonly used when the contract state needs to be modified, such as incrementing a counter or updating a setting.

```

Copy /// update will load the data, perform the specified action, and store the result /// in the database. This is shorthand for some common sequences, which may be useful /// /// This is the least stable of the APIs, and definitely needs some usage pubfnupdate(&mutself, action:A)->Result where A:FnOnce(T)->Result, E:From, { letinput=self.load()?; letoutput=action(input)?; self.save(&output)?; Ok(output) }

```

- Purpose
- : It allows for atomic updates to the state by accepting a closure (a function passed as an argument) that modifies the data and then saves it back to storage.
- Usage
- :update
- is ideal for changing existing data in the state, like incrementing a value, while ensuring that the storage operation is performed safely in one transaction.

Summary

- save
- : Persists or updates state in the contract’s storage.
- load
- : Retrieves and reads existing data from the contract’s storage.
- may_load
- : Fetches and deserializes data if it exists, returningOk(None)
- if no data is found at the key, and an error if deserialization fails.
- remove
- : Deletes the data associated with a specific key from storage.
- update
- : Modifies existing data in storage, allowing for atomic and safe updates.

These wrapper functions provide clean abstractions to manage the contract's state, making the contract logic easier to write and maintain. They are especially useful in ensuring that storage operations (which are critical for on-chain data) are handled efficiently and consistently across the contract. Previous Prefixed Storage Next Keymap Last updated1 month ago