how-storage-works)

Was this helpful? Edit on GitHub Export as PDF

Storage

An explainer on the varying storage frameworks for Secret contracts

How Storage Works

CosmWasm storage 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. Developers 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).

Prefixed Storage

One common technique in smart contracts, especially when multiple types of data are being stored, is to create separate sub-stores with unique prefixes. Thus instead of directly dealing with storage, we wrap it and put allFoo in a Storage with key"foo" + id , and allBar in a Storage with key"bar" + id . This lets us add multiple types of objects without too much cognitive overhead. Similar separation like Mongo collections or SQL tables.

Since we have different types forStorage andReadonlyStorage , we use two different constructors:

```

Copy usecosmwasm_std::testing::MockStorage; usecosmwasm_storage::{prefixed, prefixed_read};

letmutstore=MockStorage::new();

letmutfoos=prefixed(b"foo",&mutstore); foos.set(b"one",b"foo");

letmutbars=prefixed(b"bar",&mutstore); bars.set(b"one",b"bar");

letread_foo=prefixed_read(b"foo",&store); assert_eq!(b"foo".to_vec(), read_foo.get(b"one").unwrap());

letread_bar=prefixed_read(b"bar",&store); assert_eq!(b"bar".to_vec(), read_bar.get(b"one").unwrap());

```

Please note that only one mutable reference to the underlying store may be valid at one point. The compiler sees we do not ever usefoos after constructingbars , so this example is valid. However, if we did usefoos again at the bottom, it would properly complain about violating unique mutable reference.

The takeaway is to create thePrefixedStorage objects when needed and not to hang around to them too long.

Typed Storage

As we divide our storage space into different subspaces or "buckets", we will quickly notice that each "bucket" works on a unique type. This leads to a lot of repeated serialization and deserialization boilerplate that can be removed. We do this by wrapping aStorage with a type-awareTypedStorage struct that provides us a higher-level access to the data.

Note thatTypedStorage itself does not implement theStorage interface, so when combining withPrefixStorage , make sure to wrap the prefix first.

```

Copy usecosmwasm_std::testing::MockStorage; usecosmwasm_storage::{prefixed, typed};

letmutstore=MockStorage::new(); letmutspace=prefixed(b"data",&mutstore); letmutbucket=typed::<_,Data>(&mutspace);

// save data letdata=Data{ name:"Maria".to_string(), age:42, }; bucket.save(b"maria",&data).unwrap();

// load it properly letloaded=bucket.load(b"maria").unwrap(); assert_eq!(data, loaded);

// loading empty can return Ok(None) or Err depending on the chosen method: assert!(bucket.load(b"john").is_err()); assert_eq!(bucket.may_load(b"john"),Ok(None));

```

Beyond the basicsave ,load , andmay_load , there is a higher-level API exposed,update .Update will load the data, apply an operation and save it again (if the operation was successful). It will also return any error that occurred, or the final state that was written if successful.

```

Copy leton_birthday=|mutm:Option|matchm { Some(mutd)=>{ d.age+=1; Ok(d) }, None=>NotFound{ kind:"Data"}.fail(), }; letoutput=bucket.update(b"maria",&on_birthday).unwrap(); letexpected=Data{ name:"Maria".to_string(), age:43, }; assert_eq!(output, expected);

```

Bucket

Since the above idiom (a subspace for a class of items) is so common and useful, and there is no easy way to return this from a function (bucket holds a reference to space, and cannot live longer than the local variable), the two are often combined into aBucket . A Bucket works just like the example above, except the creation can be in another function:

```

Copy usecosmwasm_std::StdResult; usecosmwasm_std::testing::MockStorage; usecosmwasm_storage::{bucket,Bucket};

fnpeople<'a,S:Storage>(storage:&'amutS)->Bucket<'a,S,Data> { bucket(b"people", storage) }

fndo_stuff()->StdResult<()> { letmutstore=MockStorage::new(); people(&mutstore).save(b"john",&Data{ name:"John", age:314, })?; OK(()) }

```

Singleton

Singleton is another wrapper around theTypedStorage API. There are cases when we don't need a whole subspace to hold arbitrary key-value lookup for typed data, but rather a single storage key. The simplest example is someconfiguration information for a contract. For example, in the name service example , there is aBucket to look up name to name data, but we also have aSingleton to store global configuration - namely the price of buying a name.

Please note that in this context, the term "singleton" does not refer tothe singleton pattern but a container for a single element.

```

Copy usecosmwasm_std::{Coin, coin,StdResult}; usecosmwasm_std::testing::MockStorage;

usecosmwasm_storage::{singleton};

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

pubstructConfig{ pubpurchase_price:Option, pubtransfer_price:Option, }

fninitialize()->StdResult<()> { letmutstore=MockStorage::new(); letconfig=singleton(&mutstore,b"config"); config.save(&Config{ purchase_price:Some(coin("5","FEE")), transfer_price:None, })?; config.update(|mutcfg|{ cfg.transfer_price=Some(coin(2,"FEE")); Ok(cfg) })?; letloaded=config.load()?; OK(()) }

```

Singleton works just likeBucket , except thesave ,load ,update methods don't take a key, andupdate requires the object to already exist, so the closure takes typeT , rather thanOption . (Usesave to create the object the first time). ForBuckets , we often don't know which keys exist, butSingleton s should be initialized when the contract is instantiated.

Since the heart of much of the smart contract code is simply transformations upon some stored state, we may be able to just code the state transitions and let theTypedStorage APIs take care of all the boilerplate.

Summary

CosmWasm storage is built on a key-value storage design, similar to a HashMap, where data is stored in binary format and accessed using storage keys represented as byte arrays (&[u8]). Developers can serialize and deserialize data to/from binary, simplifying this process through wrapper functions. Storage keys are typically constants declared in state files and are not meant to be secret.

Prefixed storage allows for creating separate sub-stores with unique prefixes for organizing different data types, while Typed Storage provides a type-aware interface to reduce serialization boilerplate. CosmWasm also supports advanced abstractions like Buckets and Singletons. Buckets create subspaces for storing collections of items, while Singletons manage single elements, typically used for global contract configuration data. These APIs streamline storage handling, enabling efficient data management in smart contracts.

Now, we will explore some of these storage methods, such asPrefixedStorage ,Singleton , andKeymap , in more depth to see how they interact with on-chain data. Previous Deps/DepsMut Next Prefixed Storage Last updated1 month ago