Migration

Migration is the process through which a given smart contracts code can be swapped out or 'upgraded'.

CosmWasm made contract migration a first-class experience. When instantiating a contract, there is an optional admin field that you can set. If it is left empty, the contract is immutable. If it is set (to an external account or governance contract), that account can trigger a migration. The admin can also change admin or even make the contract fully immutable after some time. However, when we wish to migrate from contract A to contract B, contract B needs to be aware somehow of how the state was encoded.

This is where CW2 specification comes in. It specifies one specialSingleton to be stored on disk by all contracts on instantiate. When the migrate function is called, then the new contract can read that data and see if this is an expected contract we can migrate from. And also contain extra version information if we support multiple migrate paths.

Working with CW2 is quite straightforward in that, as a smart contract developer you need only perform a couple of steps.

The CW2 Spec provides aset_contract_version which should be used in instantiate to store the original version of a contract. It is important to alsoset_contract_version as a part of thepub fn migrate(...) logic this time (as opposed toinstantiate) for the contract version to be updated after a succesful migration.

const

CONTRACT_NAME :

& str

=

env! ("CARGO_PKG_NAME") ; const

CONTRACT_VERSION :

& str

=

env! ("CARGO_PKG_VERSION") ;

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

instantiate (deps :

DepsMut , env :

Env , info :

MessageInfo , msg :

InstantiateMsg)

->

Response

{ // Use CW2 to set the contract version, this is needed for migrations set_contract_version (deps . storage ,

CONTRACT_NAME ,

CONTRACT_VERSION) ? ; } Additionally,get_contract_version is provided in CW2 which can and should be used in themigrate function of the contract when you need to know the previous version of the contract. Both methods work on aItem data structure fromcw_storage_plus which operates over this object:

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

pub

struct

ContractVersion

{ /// contract is the crate name of the implementing contract, eg.crate:cw20-base /// we will use other prefixes for other languages, and their standard global namespacing pub contract :

String , /// version is any string that this implementation knows. It may be simple counter "1", "2". /// or semantic version on release tags "v0.7.0", or some custom feature flag list. /// the only code that needs to understand the version parsing is code that knows how to /// migrate from the given contract (and is tied to it's implementation somehow) pub version :

String , }

Setting up a contract for migrations

Performing a contract migration is a three step process. First, you must write a newer version of the contract you wish to update. Second, you can upload the new code as you did before, but don’t instantiate it. Third, you use a dedicatedMsgMigrateContract transaction to point this contract to use the new code. And you are done!

During the migration process, the migrate function defined on the new code is executed, never the one from the old code. Both the source and the destinationcode_id 's may be migratable, but it is necessary that the new code has amigrate function defined and properly exported as anentry_point : #[cfg_attr(not(feature = "library"), entry_point)].

Themigrate function itself, exposes the ability to make any granular changes needed to the State, akin to a database migration or any other things you might want to do.

If the migrate function returns an error, the transaction aborts, all state changes are reverted and the migration is not performed.

Provided below are a few variants on migrations you could do ranging from a very simple one, to a more restricted one by code iD and type.

Basic Contract Migration

This migration will be the most common you may see. It simply is used to swap out the code of a contract. Safety checks may not be performed if you do not also usecw2::set_contract_version .

const

CONTRACT_NAME :

& str

=

env! ("CARGO_PKG_NAME") ; const

CONTRACT_VERSION :

& str

=

env! ("CARGO_PKG_VERSION") ;

[entry_point]

pub

fn

migrate (deps :

DepsMut , _env :

Env , msg :

MigrateMsg)

->

Result < Response ,

ContractError

{ // No state migrations performed, just returned a Response Ok (Response :: default ( )) }

Restricted Migration by code version and name

This migration is a more complete and restricted example where thecw2 package is used and themigrate function ensures that:

- We are migrating from the same type of contract; checking its name
- We are upgrading from an older version of the contract; checking its version

const

CONTRACT_NAME :

& str

=

env! ("CARGO_PKG_NAME") ; const

CONTRACT_VERSION :

& str

=

env! ("CARGO_PKG_VERSION") ;

[entry_point]

pub

fn

migrate (deps :

DepsMut , _env :

Env , msg :

MigrateMsg)

->

Result < Response ,

ContractError

{ let ver =

cw2 :: get_contract_version (deps . storage) ? ; // ensure we are migrating from an allowed contract if ver . contract !=

CONTRACT_NAME

{ return

Err (StdError :: generic_err ("Can only upgrade from same type") . into ( )) ; } // note: better to do proper semver compare, but string compare *usually* works

[allow(clippy::cmp_owned)]

if ver . version

=

CONTRACT_VERSION

{ return

Err (StdError :: generic_err ("Cannot upgrade from a newer version") . into ( )) ; }

// set the new version cw2 :: set_contract_version (deps . storage ,

CONTRACT_NAME ,

CONTRACT_VERSION) ? ;

// do any desired state migrations...

Ok (Response :: default ( )) }

Migrate which updates the version only if newer

This migration is a less restrictive example than above. In this case thecw2 package is used and themigrate function ensures that:

- If the contract version has incremented from the stored one, perform needed migrations but store the new contract version
- Uses Semver instead of String compare

const

CONTRACT_NAME :

& str

=

env! ("CARGO_PKG_NAME") ; const

CONTRACT_VERSION :

& str

=

env! ("CARGO_PKG_VERSION") ;

[entry_point]

pub

fn

migrate (deps :

DepsMut , _env :

Env , msg :

MigrateMsg)

->

Result < Response ,

ContractError

{ let version :

Version

=

CONTRACT_VERSION . parse ( ) ? ; let storage_version :

Version

=

get_contract_version (deps . storage) ? . version . parse ( ) ? ;

if storage_version < version { set_contract_version (deps . storage ,

CONTRACT_NAME ,

CONTRACT_VERSION) ? ;

// If state structure changed in any contract version in the way migration is needed, it // should occur here } Ok (Response :: default ( )) } This example uses Semver to help with version checks, you would also need to add the semver dependency to your cargo deps:

[dependencies] semver = "1" And also implement Semver custom errors for your contract package:

[derive(Error, Debug, PartialEq)]

pub

enum

ContractError

{

[error(

"Semver parsing error: {0}")] SemVer (String) ,

} impl

From < semver :: Error

for

ContractError

{ fn

from (err :

semver :: Error)

->

Self

{ Self :: SemVer (err . to_string ( )) } }

Using migrate to update otherwise immutable state

This example shows how a migration can be used to update a value that generally should not be changed. This allows for the immutable value to be changed only during a migration if that functionality is needed by your team.

[entry_point]

pub

fn

migrate (deps :

DepsMut , _env :

Env , msg :

MigrateMsg)

->

Result < Response ,

HackError

{ let data = deps . storage . get (CONFIG_KEY) . ok_or_else (| |

StdError :: not_found ("State")) ? ; let

mut config :

State

=

from_slice (& data) ? ; config . verifier = deps . api . addr_validate (& msg . verifier) ? ; deps . storage . set (CONFIG_KEY ,

& to_vec (& config) ?) ;

Ok (Response :: default ( )) } In the above example, ourMigrateMsg has averifier field which contains the new value for our contractsverifier field located in the State. Provided your contract does not also expose anUpdateState or something likeUpdateVerifier ExecuteMsgs then this provides the only method to change theverifier value.

Using migrations to 'burn' a contract

Migrations can also be used to completely abandon an old contract and burn its state. This has varying uses but in the event you need it you can find an examplehere :

[entry_point]

pub

fn

migrate (deps :

DepsMut , env :

Env , msg :

MigrateMsg)

->

StdResult < Response

{ // delete all state let keys :

Vec < _

= deps . storage . range (None ,

None ,

Order :: Ascending) . map (| (k , _) | k) . collect ( ) ; let count = keys . len ( ) ; for k in keys { deps . storage . remove (& k) ; }

// get balance and send all to recipient let balance = deps . querier . query_all_balances (env . contract . address) ? ; let send =

BankMsg :: Send

{ to_address : msg . payout . clone ( ) , amount : balance , } ;

let data_msg =

format! ("burnt {} keys" , count) . into_bytes ( ) ;

Ok (Response :: new ( ) . add_message (send) . add_attribute ("action" ,

"burn") . add_attribute ("payout" , msg . payout) . set_data (data_msg)) } In the above example, when the migration occurs the State is completely deleted. Additionally all the balance of contract is send to a nominatedpayout address provided in theMigrationMsg . In this case all funds are drained and all state removed effectively burning the contract.

Platform Specific Variations

Different chains and hubs in the Cosmos ecosystem may have some variations on how migrations are done on their respective networks. This section will attempt to outline those.

Terra

Terra has some specific differences in how they manage migrations. Firstly; the contract must have been set as migratable on instantiation. The contract needs to have an admin for migratability similar to the standard procedure for migrations. Specifically migration in this case for Terra refers to swapping out the code id for a new one that is considered 'compatible' (CW2 helps with this).Source .

Note: In Terra, it is also possible to migrate a code_id across chains (COL4->5 for example). This operation is atomic and can only be performed once. Its intention is to migrate your code to the new chain and preserve its old code ID. This process helps to prevent downstream breakages of other contracts on the new network which depend on your old code ID. Example command for migrating across chains :

terrad tx wasm store . / { code . wasm }

-

from { keyname } \ - - migrate - code - id { codeID Previous Verifying Smart Contracts Next Code Pinning * Setting up a contract for migrations * * Basic Contract Migration * * Restricted Migration by code version and name * * Migrate which updates the version only if newer * * Using migrate to update otherwise immutable state * * Using migrations to 'burn' a contract * Platform Specific Variations * * Terra