Contract Composition

Given theActor model of dispatching messages, andsynchronous queries implemented in CosmWasm v0.8, we have all the raw components to enable arbitrary composition of contracts with both other contracts and native modules. Here we will explain how the components fit together and how they can be extended.

Terminology

For the remainder of this article, I will make a key distinction between "Contracts" and "Native Modules". "Contracts" are CosmWasm code that is dynamically uploaded to the blockchain at a given address. This can be added after the fact and is not tied to any runtime implementation. "Native Modules" are basically Go Cosmos SDK modules, which are compiled into the blockchain binary. These are relatively static (requiring a soft or hard fork to be added or modified) and will differ between different blockchains running CosmWasm.

We support composition between both types, but we must look more deeply at the integration with "Native Modules", as using those can causePortability issues. To minimize this issue, we provide some abstractions around "Modules"

Messages

Bothinit andhandle can return an arbitrary number ofCosmosMsg objects, which will be re-dispatched in the same transaction (thus providing atomic success/rollback with the contract execution). There are 3 classes of messages:

- Contract
-
    - This will call a given contract address with a given message (provided in serialized form). It assumes
- the caller has access to the API format.
- Module interfaces
-
    - These are standardized interfaces that can be supported by any chain to expose native
- modules under aportable
- interface.
- Custom
-
    - This encapsulates a chain-dependent extension to the message types to call into custom
- native modules. Ideally, they should beimmutable
- on the same chain over time, but they make no portability
- guarantees.

Queries

As of CosmWasm v0.8, we now allow the contracts to make synchronousread-only queries to the surrounding runtime. As withMessages , we have three fundamental types:

- Contract
-
    - This will query a given contract address with a given message (provided in serialized form). It assumes
- the caller has access to the API format.
- Module interfaces
-
    - These are standardized interfaces that can be supported by any chain to expose native
- modules under aportable
- interface.
- Custom
-
    - This encapsulates a chain-dependent extension to query custom native modules. Ideally,
- they should beimmutable
- on the same chain over time, but they make no portability guarantees.

Cross-Contract queries take the address of the contract and a serializedQueryMsg in the contract-specific format, and synchronously get a binary serialized return value in the contract-specific format. It is up to the calling contract to understand the appropriate formats. To simplify this, we can provide some contract-specific type-safe wrappers, much in the way we provide a simplequery_balance method as a wrapper around thequery implementation provided by the Trait.

Modules

To enable better integrations with the native blockchain, we are providing a set of standardized module interfaces that

should work consistently with all CosmWasm chains. The most basic one is theBank module, which provides access to the underlying native tokens. This gives usBankMsg::Send as well asBankQuery::Balance andBankQuery::AllBalances to check balances and move tokens.

The second standardized module isstaking , which provides some standardized messages forDelegate ,Undelegate ,Redelegate andWithdraw , as well as queryingValidators andDelegations . These interfaces are designed to support most PoS systems and can be supported on any PoS system, not just with the current staking module of the Cosmos SDK. (So, if you fork that for your chain, the contracts should still work).

This provides a clean design, where one can develop a contract that eg. issues staking derivatives using thestaking module interface, and have confidence that that same contract will run on two different blockchains, even if they are both heavily customized and one is on Cosmos SDK 0.38 and the other on Cosmos SDK 0.39 (with many breaking changes). The downside here is that every module interface must be added to all layers of the stack, which provides some delay to supporting custom features, and we cannot easily add support for every custom module in every Cosmos SDK-based blockchain.

That said, we highly recommend using the module interface when possible and using custom types as a temporary measure to keep a fast release cycle. We are happy to work with projects with concrete use cases that can be reused between many different chains to add new standardized module interfaces. Perhaps for governance, or for IBC/light client-related features.

Note, that theoretically these module interfaces can also be implemented by contracts, not just native code. Eg. if we made a standardswap interface, and your blockchain has no nativeswap module, we could upload a UniSwap-inspired contract and register that somehow with the Go blockchain. Then the blockchain would know to take the swap message and dispatch it to this custom contract. (Note that this is not implemented at all, just an idea of future directions).

Customization

Many chains will want to allow contracts to execute with their custom go modules, without going through all the work to try to turn it into a standardizedModule Interface and wait for a new CosmWasm release. For this case, we introduce theCustom variant on bothCosmosMsg andQueryRequest . The idea here is that the contract can define the type to be included in theCustom variants, and it just needs to be agreed upon by the Cosmos SDK app (in Golang) on the other side. All the code between the contract and the blockchain codec treats it as opaque JSON bytes.

The demo "reflect" contract in the standard CosmWasm reposhows how to useCustomMsg andCustomQuery. You can see how the contractuses aCustomQuery to call out to some "runtime-provided" code. For unit tests, we canmock out the runtime querier , but in a deployed system, this should be provided by native Go code in your Cosmos SDK application.

Beyond trivial cases, we are working with Terra to expose theirswap ,oracle , andtreasury modules to CosmWasm contracts on their chains. These features need to be exposed in an immutable format that will work forever on their chain, but there is no need for portability. More interesting is the ability to quickly expose new features of their native modules to contracts on their chain. In fact, this concrete use case inspired all the refactoring to make sure custom messages and queries are possible.

Design Considerations

In producing a solid design, we want the API to fulfill all these requirements (or strike a good balance if truly impossible to achieve them all):

Portability

The same contract should run on two distinct blockchains, with differing Golang modules, different versions of the Cosmos SDK, or ideally, even based on different frameworks (eg. running on Substrate). This should be possible when avoidingCustom messages, and checking the optional features one uses. The features are currentlystaking , which assumes a PoS system, anditerator , which assumes we can do prefix scans over the storage (ie. it is a MerkleTree , not a MerkleTrie).

Immutability

Contracts are immutable and encode the query and message formats in their bytecode. If we allowed dispatchingsdk.Msg in the native format (be it json, amino, or protobuf), and the format of native messages changes, then the contracts would break. This may mean that a staking module could never undelegate the tokens. If you think this is a theoretical issue, please note that every major update of the Cosmos SDK has produced such breaking changes and has migrations for them. Migrations that cannot be performed on immutable contracts. Thus, we need to ensure that our design provides an immutable API to a potentially mutable runtime, which is a primary design criterion when designing the standardmodule interfaces .

Extensibility

We should be able to add new interfaces to a contract and blockchain without needing to update any of the intermediate layers. That is if you are building a customsuperd blockchain app, which importsx/wasm fromwasmd , and want to develop contracts on it that call into your customsuperd modules, then in an ideal world, you could add those message types to an additionalcw-superd library that you can import in your contracts and add the callbacks to them insuperd Go code.Without any changes incosmwasm-std ,cosmwasm-vm ,go-cosmwasm , orwasmd repositories (which have a different release cycle than your app).

We provide theCustom variants toCosmosMsg andQueryRequest to enable such cases. They can provide immutability but not portability.

Usability

We also want to make it not just secure andpossible to compose contracts into a larger whole, but make it simple from the developer's perspective. This applies to both the contract authors, as well as the blockchain developers integrating CosmWasm into their custom blockchain. And we want to make it easy to build client-side applications using the contracts.

We are using JSON encoding for the CosmWasm messages to make this simple and exportJSON schemas for every contract to allow auto-generation of client-side codecs. We also provideCosmJS as an easy-to-use TypeScript client library allowing access to all contracts (and bank module) on a CosmWasm-based chain.

Checking for Support

If we want to call some extensions, say to theStaking modules, we can compile our contract to handle that. But how do we detect if the blockchain can support it? We want to fail on upload or instantiation of a contract, and not discover some key functionality that doesn't work on this chain when there is value stored in the contract.

The design decision was to you use feature flags, exposed as wasm export functions, to configure which extra features are required by the contract. This lets the host chain inspect compatibility before allowing an upload. To do so, we make some "ghost" exports likerequires_staking() orrequires_terra() . This would only pass compatibility check if the runtime also exposed such features. When instantiatingx/wasm.NewKeeper() you can specify which features are supported.

Type-Safe Wrappers

When querying or calling into other contracts, we give up all the type-checks we get with native module interfaces. They require the caller to know the details of the caller. This is the same as Ethereum. However, we can provide some " interface" wrappers that a contract could export, such that other contracts can easily call into it.

For example:

pub

struct

NameService (CanonicalAddr) ;

impl

NameService

{ pub

fn

query_name (deps :

& Extern , name :

& str)

->

CanonicalAddr

{

/ .. /

} pub

fn

register (api :

& Api , name :

& str)

->

CosmosMsg

{

/ .. /

} } Rather than storing just theCanonicalAddr of the other contract in our configuration, we could storeNameService , which is a zero-cost"newtype" over the original address, with the same serialization format. However, it would provide us some type-safe helpers to make queries against the contract, as well as produceCosmosMsg for registration.

Note that these type-safe wrappers are not tied to animplementation of a contract, but rather the contract'sinterface . Thus, we could create a small library with a list of standard/popular interfaces (like the ERCxxx specs) represented with such "newtypes". A contract creator could import one of these wrappers and then easily call the contract, regardless of implementation, as long as it supported the proper interface. Previous Serialization Next Comparison with Solidity Contracts * Terminology * Messages * Queries * Modules * Customization * Design Considerations * * Portability * * Immutability * * Extensibility * Usability * * Checking for Support * * Type-Safe Wrappers