Integration

If you want to use Wasm in your own app, here is how you can get this working quickly and easily. First, check to make sure you fit the pre-requisites, then integrate thex/wasm module as described below, and finally, you can add custom messages and queries to your custom Go/SDK modules, exposing them to any chain-specific contract.

Prerequisites

The pre-requisites of integratingx/wasm into your custom app is to be using a compatible version of the Cosmos SDK, and to accept some limits to the hardware it runs on.

wasmd Cosmos SDK v0.24 v0.45.0 v0.23 v0.45.0 v0.22 v0.45.0 v0.21 v0.42.x v0.20 v0.42.x v0.19 v0.42.x v0.18 v0.42.x v0.17 v0.42.x v0.16 v0.42.x v0.15 v0.41.x v0.14 v0.40.x v0.13 v0.40.0-rc3 v0.12 v0.40.0-rc3 v0.11 v0.39.1 v0.10 v0.39.1 v0.9 v0.38.3 v0.8 v0.38.3 v0.7 v0.38.3 We currently only support Intel/AMD64 CPUs and OSX or Linux. For Linux, the standard build commands work forglibc systems (Ubuntu, Debian, CentOS, etc). If you wish to compile for amuslc based system (like alpine), you need to compile a static library wasmvm locally and compile go with themuslc build tag. Or just use theDockerfile , which builds a static go binary in an alpine system.

This limit comes from the Rust dll we use to run the wasm code, which comes fromwasmvm . There are open issues for addingARM support , and addingWindows support . However, these issues are not high on the roadmap and unless you are championing them, please count on the current limits for the near future.

Quick Trial

The simplest way to try out CosmWasm is simply to runwasmd out of the box, and focus on writing, uploading, and using your custom contracts. There is plenty that can be done there, and lots to learn.

Once you are happy with it and want to use a custom Cosmos SDK app, you may consider simply forkingwasmd .I highly advise against this . You should try one of the methods below.

Integrating wasmd

As an external module

The simplest way to usewasmd is just to importx/wasm and wire it up inapp.go . You now have access to the whole module and your custom modules running side by side. (But the CosmWasm contracts will only have access tobank andstaking ... more below oncustomization).

The requirement here is that you have imported the standard SDK modules from the Cosmos SDK, and enabled them inapp.go . If so, you can just look atwasmd/app/app.go for how to do so (just search there for lines withwasm).

wasmd also comes with a customante handler that adds the TX position in the block into the context and passes it to the contracts. In order to support this feature you would need to add our custom ante handler into theante handler chain as in:app/ante.go

Copied into your app

Sometimes, however, you will need to copyx/wasm into your app. This should be in limited cases and makes upgrading more difficult, so please take the above path if possible. This is required if you have either disabled some key SDK modules in your app (eg. using PoA not staking and need to disable those callbacks and feature support), or if you have copied in the corex/* modules from the Cosmos SDK into your application and customized them somehow.

In either case, your best approach is to copy thex/wasm module from the latest release into your application. Your goal is to makeminimal changes in this module, and rather add your customizations in a separate module. This is due to the fact that you will have to copy and customizex/wasm from upstream on all futurewasmd releases, and this should be as simple as possible.

If, for example, you have forked the standard SDK libs, you just want to change the imports (from eg.github.com/cosmos/cosmos-sdk/x/bank togithub.com/YOUR/APP/x/bank), and adjust any calls if there are compiler errors due to differing APIs (maybe you use Decimals not Ints for currencies?).

By the end of this, you should be able to run the standard CosmWasm contracts in your application, alongside all your custom logic.

Adding custom hooks

Once you have gotten this integration working and are happy with the flexibility it offers you, you will probably start wishing for deeper integration with your custom SDK modules. "It sure is nice to have custom tokens with a bonding curve from my native token, but I would love to trade them on the exchange I wrote as a Go module. Or maybe use them to add options to the exchange."

At this point, you need to dig down deeper and see how you can add this power without forking either CosmWasm orwasmd .

Calling contracts from native code

This is perhaps the easiest part. Let's say your native exchange module wants to call into a token that lives as a CosmWasm module. You need to pass thewasm.Keeper into yourexchange.Keeper . If you know the format for sending messages and querying the contract (exported as JSON schema from each contract), and have a way of configuring addresses of supported token contracts, your code can simply callwasm.Keeper.Execute with a properly formatted message to move funds, orwasm.Keeper.QuerySmart to query the contract instance via it's interface orwasm.Keeper.QueryRaw to query storage directly.

If you look at the unit tests inx/wasm/keeper , it should be pretty straightforward.

Extending the Contract Interface

If you want to let the contracts access your native modules, the first step is to define a set of Messages and Queries that you want to expose, and then add them asCosmosMsg::Custom andQueryRequest::Custom variants. You can see an example of thebindings for Terra .

Once you have those bindings, use them to build asimple contact using much of the API . Don't worry too much about the details, this should be usable, but mainly you will want to upload it to your chain and use it for integration tests with your native Cosmos SDK modules. Once that is solid, then add more and more complex contracts.

You will then likely want to add amocks package so you can provide mocks for the functionality of your native modules when unit testing the contracts (provide static data for exchange rates when your contracts query it). You can see an example ofmocks for Terra contracts .

What these three steps provide is basically a chain-specific extension to the CosmWasm contract SDK. Any CosmWasm contract can import your library (bindings and mocks) and easily get started using your custom, chain-specific extensions just as easily as using the standard CosmWasm interfaces. What is left is actually wiring them up in your chain so they work as desired.

Note, in order to ensure that no one tries to run the contracts on an unsupported chain, you will want to include arequires_XYZ directive in yourbindings library, this will mean that only blockchain apps that explicitly declare their support for theXYZ extensions (please rename XYZ to your project name) will allow the contract to be uploaded, and others get error messages upon upload, not while running a critical feature later on. You just need to adda line like this to your binding library to add the requirement to any contract that imports yourbindings lib.

Calling into the SDK

Before I show how this works, I want to remind you, that if you have copiedx/wasm , pleasedo not make these changes tox/wasm .

We will add a new module, eg.x/contracts , that will contain custom bindings between CosmWasm contracts and your native modules. There are two entry points for you to use. The first isCustomQuerier , which allows you to handle your custom queries. The second isCustomEncoder which allows you to convert theCosmosMsg::Custom(YourMessage) types to[]sdk.Msg to be dispatched.

Writing stubs for these is rather simple. You can look at thereflect_test.go file to see this in action. In particular, herewe define aCustomQuerier , and herewe define aCustomHandler . This code is responsible to takejson.RawMessage from the raw bytes serialized from your custom types in rust and parsing it into Go structs. Then take these go structs and properly convert them for your custom SDK modules.

You can look at the implementations for thestaking module to see how to build these for non-trivial cases, including passing in theKeeper via a closure. Here weencode staking messages . Note that withdraw returns 2 messages, which is an option you can use if needed to translate into native messages. When wehandle staking queries we take in aKeeper in the closure and dispatch the customQueryRequest from the contract to the nativeKeeper interface, then encodes a response. When defining the return types, note that for proper parsing in the Rust contract, you should properly name the JSON fields and use theomitempty keyword if Rust expectsOption . You must also useomitempty and pointers for all fields that correspond to a Rustenum , so exactly one field is serialized.

Wiring it all together

Once you have written and tested these custom callbacks for your module, you need to enable them in your application. The first step is to write an integration test with a contract compiled with your custom SDK to ensure it works properly, then you need to configure this inapp.go .

For the test cases, you must define the supported feature set to include your custom name (rememberrequires_XYZ above?). Then, when creatingTestInput , you can pass in your custom encoder and querier . Run a few tests with your compiled contract, ideally exercising the majority of the interfaces to ensure that all parsing between the contract and the SDK is implemented properly.

Once you have tested this and are happy with the results, you can wire it up inapp.go . Just edit the defaultNewKeeper constructor to have the propersupportedFeatures and pass in theCustomEncoder andCustomQuerier as the last two arguments toNewKeeper . Now you can compile your chain and upload your custom contracts on it. Previous Sudo Execution Next Idea behind an Actor Model * Prerequisites * Quick Trial * Integrating wasmd * * As an external module * * Copied into your app * Adding custom hooks * * Calling contracts from native code * * Extending the Contract Interface * * Calling into the SDK * * Wiring it all together