| **SMART CONTRACT** | <https://github.com/ivmidable/deposit-native-example> |
| --- | --- |

**Comments in blue.**

**Code Changes in green.**

**Knowledge unanswered in red.**

| **CODE REVIEW** |
| --- |

**State.rs**

// derive - implements the following traits in the struct Config.

// Serialize/Deserialize to JSON schema, Clone (copy), Debug (can be formatted with the debug trait :?),

// PartialEq - Partial equivalence relations (e.g. we can compare with only one struct field), JsonSchema

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

pub struct Config {

pub owner: Addr

}

// Struct that is going to be stored in cw\_storage\_plus DEPOSITS

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

pub struct Deposits {

pub count: i32,

pub owner: Addr,

pub coins: Coin

}

//Map using a tuple as key, and the Struct Deposits as value DEPOSITS: Map<(&address, &coin\_denom), Deposit>

//key is address, denom

pub const DEPOSITS: Map<(&str, &str), Deposits> = Map::new("deposits");

pub const CONFIG: Item<Config> = Item::new("config");

**Msg.rs**

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

#[serde(rename\_all = "snake\_case")]

pub struct InstantiateMsg {

}

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

#[serde(rename\_all = "snake\_case")]

pub enum ExecuteMsg {

Deposit { },

Withdraw { amount:u128, denom:String },

}

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

#[serde(rename\_all = "snake\_case")]

pub enum QueryMsg {

Deposits { address: String },

GetConfig {},

}

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

#[serde(rename\_all = "snake\_case")]

pub struct DepositResponse {

pub deposits: Vec<(String, Deposits)>,

}

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

#[serde(rename\_all = "snake\_case")]

pub enum MigrateMsg {}

// InstantiateMsg - enum

// ExecuteMsg/QueryMsg - enum of structs.

// DepositRespose - struct of a tuple (String, Deposits Struct) vector

**Contract.rs**

// Conditional compilation, making sure that the entry point is not created more than once

#[cfg\_attr(not(feature = "library"), entry\_point)]

pub fn instantiate(

\_deps: DepsMut,

\_env: Env,

\_info: MessageInfo,

\_msg: InstantiateMsg,

) -> Result<Response, ContractError> {

//When we instantiate the contract, the caller becomes the owner. Stored in the State/Config

CONFIG.save(

\_deps.storage,

&Config {

owner: \_info.sender.clone(),

},

)?;

Ok(Response::default())

}

#[cfg\_attr(not(feature = "library"), entry\_point)]

pub fn execute(

deps: DepsMut,

\_env: Env,

info: MessageInfo,

msg: ExecuteMsg,

) -> Result<Response, ContractError> {

match msg {

ExecuteMsg::Deposit {} => execute::execute\_deposit(deps, info),

ExecuteMsg::Withdraw { amount, denom } => {

execute::execute\_withdraw(deps, info, amount, denom)

}

}

}

#[cfg\_attr(not(feature = "library"), entry\_point)]

pub fn query(deps: Deps, \_env: Env, msg: QueryMsg) -> StdResult<Binary> {

match msg {

QueryMsg::Deposits { address } => to\_binary(&query::query\_deposits(deps, address)?),

QueryMsg::GetConfig {} => to\_binary(&query::get\_config(deps)?),

}

}

pub mod execute {

use super::\*;

pub fn execute\_deposit(deps: DepsMut, info: MessageInfo) -> Result<Response, ContractError> {

let sender = info.sender.clone().into\_string();

// Only one fund transfer

if info.funds.len() != 1 {

return Err(ContractError::OnlyOneCoin {});

}

// d\_coins represents the first element of the funds vector. <Coin, Global>

// Coin : {denom, amount} . Global : Empty Struct

let d\_coins = info.funds[0].clone();

let config = CONFIG.load(deps.storage)?;

// Only the sender can send funds

if config.owner != info.sender {

return Err(ContractError::InvalidOwner {});

}

//check to see if deposit exists. Very neat to work with the Result returned from .load()

match DEPOSITS.load(deps.storage, (&sender, d\_coins.denom.as\_str())) {

// The Map structure stores a deposit. If that coin has already been deposited its amount is increased.

// Important to remember to use checked\_add to avoid many kinds of possible errors. Never += 1

Ok(mut deposit) => {

//add coins to their account

deposit.coins.amount = deposit.coins.amount.checked\_add(d\_coins.amount).unwrap();

deposit.count = deposit.count.checked\_add(1).unwrap();

DEPOSITS

.save(deps.storage, (&sender, d\_coins.denom.as\_str()), &deposit)

.unwrap();

}

Err(\_) => {

//user does not exist, add them. Create the deposit and store it in the map

let deposit = Deposits {

count: 1,

owner: info.sender,

coins: d\_coins.clone(),

};

DEPOSITS

.save(deps.storage, (&sender, d\_coins.denom.as\_str()), &deposit)

.unwrap();

}

}

Ok(Response::new()

.add\_attribute("execute", "deposit")

.add\_attribute("denom", d\_coins.denom)

.add\_attribute("amount", d\_coins.amount))

}

pub fn execute\_withdraw(

deps: DepsMut,

info: MessageInfo,

amount: u128,

denom: String,

) -> Result<Response, ContractError> {

let sender = info.sender.clone().into\_string();

// QUESTION: Shouldn't we check that the owner is the one withdrawing?

// **CODE ADDED**

let config = CONFIG.load(deps.storage)?;

// Only the sender can withdraw funds

if config.owner != info.sender {

return Err(ContractError::InvalidOwner {});

}

// Here we are loading the Deposits the sender has for a certain kind of funds.

// QUESTION: What happens if there are not that kind of funds?. may\_load vs match DEPOSITS.load()

// Alternative code suggested in GENERAL QUESTIONS (question 4)

let mut deposit = DEPOSITS

.load(deps.storage, (&sender, denom.as\_str()))

.unwrap();

// The amount and count is reduced, using checked\_sub() to subtract efficiently.

deposit.coins.amount = deposit

.coins

.amount

.checked\_sub(Uint128::from(amount))

.unwrap();

deposit.count = deposit.count.checked\_sub(1).unwrap();

DEPOSITS

.save(deps.storage, (&sender, denom.as\_str()), &deposit)

.unwrap();

// As we have reduced the amount of funds from our records we send the funds back to the sender.

let msg = BankMsg::Send {

to\_address: sender.clone(),

amount: vec![coin(amount, denom.clone())],

};

Ok(Response::new()

.add\_attribute("execute", "withdraw")

.add\_attribute("denom", denom)

.add\_attribute("amount", amount.to\_string())

.add\_message(msg))

}

pub fn update\_config(

deps: DepsMut,

info: MessageInfo,

owner: Option<String>,

) -> Result<Response, ContractError> {

// The owner parameter is an Option. This way if the parameter is None, you load and save the same info

//but if the parameter is Some() you update the config.owner

let mut config = CONFIG.load(deps.storage)?;

if config.owner != info.sender {

return Err(ContractError::InvalidOwner {});

}

if let Some(owner) = owner {

config.owner = deps.api.addr\_validate(&owner)?;

}

CONFIG.save(deps.storage, &config)?;

Ok(Response::default())

}

}

pub mod query {

use super::\*;

pub fn get\_config(deps: Deps) -> StdResult<Config> {

let config = CONFIG.load(deps.storage)?;

Ok(config)

}

// Query the deposits transferred by a certain address.

// YET TO LEARN: Not sure how prefix works on Map<(&address, &coin\_denom), Deposit>

pub fn query\_deposits(deps: Deps, address: String) -> StdResult<DepositResponse> {

let res: StdResult<Vec<\_>> = DEPOSITS

.prefix(&address)

.range(deps.storage, None, None, Order::Ascending)

.collect();

let deposits = res?;

Ok(DepositResponse { deposits })

}

}

#[cfg(test)]

mod tests {

use super::\*;

use cosmwasm\_std::testing::{mock\_dependencies, mock\_env, mock\_info};

use cosmwasm\_std::{coin, from\_binary};

const SENDER: &str = "sender\_address";

const AMOUNT: u128 = 100000;

const DENOM: &str = "utest";

fn setup\_contract(deps: DepsMut) {

let msg = InstantiateMsg {};

let info = mock\_info(SENDER, &[]);

let res = instantiate(deps, mock\_env(), info, msg).unwrap();

assert\_eq!(0, res.messages.len());

}

fn deposit\_coins(deps: DepsMut) {

let msg = ExecuteMsg::Deposit {};

let coins = vec![coin(AMOUNT, DENOM.to\_string())];

let info = mock\_info(SENDER, &coins);

let res = execute(deps, mock\_env(), info, msg).unwrap();

assert\_eq!("deposit".to\_string(), res.attributes[0].value);

assert\_eq!(DENOM.to\_string(), res.attributes[1].value);

assert\_eq!(AMOUNT.to\_string(), res.attributes[2].value);

}

fn withdraw\_coins(deps: DepsMut) {}

fn query\_coins(deps: Deps) {

let msg: QueryMsg = QueryMsg::Deposits {

address: SENDER.to\_string(),

};

let res = query(deps, mock\_env(), msg).unwrap();

let query = from\_binary::<DepositResponse>(&res).unwrap();

// query : DepositResponse { pub deposits: Vec<(String, Deposits)>,}

// pub struct Deposits { pub count: i32, pub owner: Addr, pub coins: Coin}

// pub struct Coin { pub denom: String, pub amount: Uint128,}

assert\_eq!(SENDER, query.deposits[0].1.owner);

assert\_eq!(DENOM, query.deposits[0].1.coins.denom);

assert\_eq!(

AMOUNT.to\_string(),

query.deposits[0].1.coins.amount.to\_string()

);

assert\_eq!(1, query.deposits[0].1.count);

}

// Have not seen before this kind of testing with functions declared previously. Neat.

#[test]

fn \_0\_instantiate() {

let mut deps = mock\_dependencies();

setup\_contract(deps.as\_mut());

}

#[test]

fn \_1\_deposit() {

let mut deps = mock\_dependencies();

setup\_contract(deps.as\_mut());

deposit\_coins(deps.as\_mut());

}

#[test]

fn \_2\_query\_deposit() {

let mut deps = mock\_dependencies();

setup\_contract(deps.as\_mut());

deposit\_coins(deps.as\_mut());

query\_coins(deps.as\_ref());

}

#[test]

fn \_1\_deposit\_then\_withdraw() {

let mut deps = mock\_dependencies();

setup\_contract(deps.as\_mut());

deposit\_coins(deps.as\_mut());

}

}

| **GENERAL QUESTIONS** |
| --- |

1. **What are the concepts (borrowing, ownership, vectors etc)**

The Concepts in the Code are Structs, Item, Map, Vectors, Tupples, Ownerships, Enum, Result, Coins…

From the basics of rust and cosmwasm, I would say most of it can be found inside this contract.

1. **What is the organization?**

Query and Execute code is written in their own module.

Testing defines a number of functions that are called by the tests themselves. This clearly reduces the code needed to write code.

1. **What is the contract doing? What is the mechanism?**

The contract is depositing and withdrawing coins.

The owner of the contract is the only one that can perform those actions.

The owner can transfer ownership to another account/address.

1. **How could it be better? More efficient? Safer?**

Withdrawing non-exiting funds it is not covered. The alternative is suggested in the next functions:

pub fn execute\_withdraw(

deps: DepsMut,

info: MessageInfo,

amount: u128,

denom: String,

) -> Result<Response, ContractError> {

let sender = info.sender.clone().into\_string();

let config = CONFIG.load(deps.storage)?;

if config.owner != info.sender {

return Err(ContractError::InvalidOwner {});

}

match DEPOSITS.load(deps.storage, (&sender, denom.as\_str())) {

Ok(mut deposit) => {

deposit.coins.amount = deposit

.coins

.amount

.checked\_sub(Uint128::from(amount))

.unwrap();

deposit.count = deposit.count.checked\_sub(1).unwrap();

DEPOSITS

.save(deps.storage, (&sender, denom.as\_str()), &deposit)

.unwrap();

let msg = BankMsg::Send {

to\_address: sender.clone(),

amount: vec![coin(amount, denom.clone())],

};

Ok(Response::new()

.add\_attribute("execute", "withdraw")

.add\_attribute("denom", denom)

.add\_attribute("amount", amount.to\_string())

.add\_message(msg))

}

Err(\_) => {

Err(ContractError::CustomError { val: "No funds for the withdrawn coin".to\_string() })

}

}