

Comments in blue.

Code Changes in green.

## 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.
  // .prefix() let's us filter by address (Partial key of DEPOSITS Map)
  // 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, Tuples, Ownerships, Enum, Result, Coins...  
From the basics of rust and cosmwasm, I would say most of it can be found inside this contract.

### 2. 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.

### 3. 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.

### 4. How could it be better? More efficient? Safer?

Withdrawing non-existing 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() })

}
}

```