SMART CONTRACT	https://github.com/InjectiveLabs/cw-injective/tree/dev/contracts/registry/src

Scope: Smart Contract

CODE REVIEW

state.rs

```
use schemars::JsonSchema;
use serde::{Deserialize, Serialize};

use cosmwasm_std::Addr;
use cw_storage_plus::{Index, IndexList, IndexedMap, MultiIndex};

// Contract struct defines begin blocker contract execution params.
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, Eq, JsonSchema)]
pub struct Contract {
    pub gas_limit: u64,
    pub gas_price: u64,
    pub is_executable: bool,
}

pub(crate) const INACTIVE_CONTRACT: u8 = 0;
pub(crate) const ACTIVE_CONTRACT: u8 = 1;

// ContractIndex struct with a single field 'active' that is a MultiIndex. Active
will be liked to the field is_executable.
pub struct ContractIndexes<'a> {
    pub active: MultiIndex<'a, u8, Contract, Addr>,
}

// ContractIndexes will index the Contract Structs by their active/is_executable
field.
// Use of prefix with ACTIVE_CONTRACT can be found in code.
impl<'a> IndexList<Contract> for ContractIndexes<'a> {
    fn get_indexes(&'_ self) -> Box<dyn Iterator<Item = &'_ dyn Index<Contract>> + '_>
{
        let v: Vec<&dyn Index<Contract>> = vec![&self.active];
        Box::new(v.into_iter())
    }
}
```

```rust
// Contract function returns an IndexedMap that stores and index the Contracts
struct.
// To index it uses a MultiIndex which relays on is_executable, although it is coded
and renamed as 'active'
// This IndexedMap uses ContractIndexes struct to index the Contract structs by their
active (is_executable) field.
// IndexedMap(Addr, Contract) with MultiIndex ContractIndexes
pub fn contracts<'a>() -> IndexedMap<'a, &'a Addr, Contract, ContractIndexes<'a>> {
    let indexes = ContractIndexes {
        active: MultiIndex::new(
            |_, c: &Contract| {
                if c.is_executable {
                    ACTIVE_CONTRACT
                } else {
                    INACTIVE_CONTRACT
                }
            },
            "contracts",
            "contracts__active",
        ),
    };
    IndexedMap::new("contracts", indexes)
}
```

message.rs

```rust
use cosmwasm_std::Addr;
use schemars::JsonSchema;
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, Eq, JsonSchema)]
pub struct InstantiateMsg {}


// ExecuteMsgs are self-explanatory based on their names
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, Eq, JsonSchema)]
#[serde(rename_all = "snake_case")]
pub enum ExecuteMsg {
    Register {
        contract_address: Addr,
        gas_limit: u64,
```

```
        gas_price: u64,
        is_executable: bool,
    },
    Deregister {
        contract_address: Addr,
    },
    Update {
        contract_address: Addr,
        gas_limit: u64,
        gas_price: u64,
    },
    Activate {
        contract_address: Addr,
    },
    Deactivate {
        contract_address: Addr,
    },
}

#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, Eq, JsonSchema)]
#[serde(rename_all = "snake_case")]
pub enum QueryMsg {
    // GetContracts returns the registered contracts as a json-encoded number
    GetContract {
        contract_address: Addr,
    },
    GetContracts {
        start_after: Option<String>,
        limit: Option<u32>,
    },
    GetActiveContracts {
        start_after: Option<String>,
        limit: Option<u32>,
    },
}

// Contract struct defines begin blocker contract execution params.
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, Eq, JsonSchema)]
pub struct ContractExecutionParams {
    pub address: Addr,
    pub gas_limit: u64,
    pub gas_price: u64,
    pub is_executable: bool,
}

// We define a custom struct for each query response
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, Eq, JsonSchema)]
pub struct ContractsResponse {
```

```
    pub contracts: Vec<ContractExecutionParams>,
}

// We define a custom struct for each query response
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, Eq, JsonSchema)]
pub struct ContractResponse {
    pub contract: ContractExecutionParams,
}
```

contract.rs

```
#[cfg(not(feature = "library"))]
use cosmwasm_std::entry_point;
use cosmwasm_std::{
    to_binary, Addr, Binary, Deps, DepsMut, Env, MessageInfo, Order, Response,
StdResult,
};
use cw2::set_contract_version;
use cw_storage_plus::Bound;
use cw_utils::maybe_addr;

use crate::error::ContractError;
use crate::msg::{
    ContractExecutionParams, ContractResponse, ContractsResponse, ExecuteMsg,
InstantiateMsg,
    QueryMsg,
};
use crate::state::{contracts, Contract, ACTIVE_CONTRACT};

// version info for migration info
const CONTRACT_NAME: &str = "crates.io:registry";
const CONTRACT_VERSION: &str = env!("CARGO_PKG_VERSION");

// On instantiation no state is recorded, only the contract version is set and report
some attributes.
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn instantiate(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    _msg: InstantiateMsg,
) -> Result<Response, ContractError> {
    set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)?;
```

```
    Ok(Response::new()
        .add_attribute("method", "instantiate")
        .add_attribute("owner", info.sender))
}
// The contract itself will be able to access this entry point. Nobody else.
// sudo does not have info as a parameter. No sender or coins passed.
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn sudo(deps: DepsMut, _env: Env, msg: ExecuteMsg) -> Result<Response,
ContractError> {
    match msg {
        ExecuteMsg::Register {
            contract_address,
            gas_limit,
            gas_price,
            is_executable,
        } => try_register(deps, contract_address, gas_limit, gas_price,
is_executable),
        ExecuteMsg::Deregister { contract_address } => try_deregister(deps,
contract_address),
        ExecuteMsg::Update {
            contract_address,
            gas_limit,
            gas_price,
        } => try_update(deps, contract_address, gas_limit, gas_price),
        ExecuteMsg::Activate { contract_address } => try_activate(deps,
contract_address),
        ExecuteMsg::Deactivate { contract_address } => try_deactivate(deps,
contract_address),
    }
}

#[cfg_attr(not(feature = "library"), entry_point)]
pub fn execute(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    msg: ExecuteMsg,
) -> Result<Response, ContractError> {
    match msg {
        ExecuteMsg::Register { .. } => Err(ContractError::Unauthorized {}),
        ExecuteMsg::Deregister { .. } => Err(ContractError::Unauthorized {}),
        ExecuteMsg::Update {
            contract_address,
            gas_limit,
            gas_price,
        } => {
            only_managed_contract(&contract_address, info)?;
```

```
            try_update(deps, contract_address, gas_limit, gas_price)
        }
        ExecuteMsg::Activate { contract_address } => {
            only_managed_contract(&contract_address, info)?;
            try_activate(deps, contract_address)
        }
        ExecuteMsg::Deactivate { contract_address } => {
            only_managed_contract(&contract_address, info)?;
            try_deactivate(deps, contract_address)
        }
    }
}

pub fn only_registry(env: Env, info: MessageInfo) -> Result<(), ContractError> {
    // Check if the sender is the registry contract address (only wasmx module can do
this)
    if env.contract.address != info.sender {
        Err(ContractError::Unauthorized {})
    } else {
        Ok(())
    }
}

// The sender must be the contract to be Updated/Activated/Deactivated
pub fn only_managed_contract(
    contract_address: &Addr,
    info: MessageInfo,
) -> Result<(), ContractError> {
    if contract_address != &info.sender {
        Err(ContractError::Unauthorized {})
    } else {
        Ok(())
    }
}

pub fn try_register(
    deps: DepsMut,
    contract_addr: Addr,
    gas_limit: u64,
    gas_price: u64,
    is_executable: bool,
) -> Result<Response, ContractError> {
    let contract = Contract {
        gas_limit,
        gas_price,
        is_executable,
    };
```

```
    // Contract can only be registered once. It uses update instead of save.
    // try to store it, fail if the address is already registered
    contracts().update(deps.storage, &contract_addr, |existing| match existing {
        None => Ok(contract),
        Some(_) => Err(ContractError::AlreadyRegistered {}),
    })?;

    let res = Response::new().add_attributes(vec![
        ("action", "register"),
        ("addr", contract_addr.as_str()),
    ]);
    Ok(res)
}

// Removes the contract_addr entry from contracts()
pub fn try_deregister(deps: DepsMut, contract_addr: Addr) -> Result<Response,
ContractError> {
    contracts().remove(deps.storage, &contract_addr)?;

    let res = Response::new().add_attributes(vec![
        ("action", "deregister"),
        ("addr", contract_addr.as_str()),
    ]);
    Ok(res)
}

pub fn try_update(
    deps: DepsMut,
    contract_addr: Addr,
    gas_limit: u64,
    gas_price: u64,
) -> Result<Response, ContractError> {
    // this fails if contract is not available
    let mut contract = contracts().load(deps.storage, &contract_addr)?;

    // update the contract
    if gas_limit != 0 {
        contract.gas_limit = gas_limit;
    }
    if gas_price != 0 {
        contract.gas_price = gas_price;
    }

    // and save
    contracts().save(deps.storage, &contract_addr, &contract)?;

    let res = Response::new()
        .add_attributes(vec![("action", "update"), ("addr", contract_addr.as_str())]);
```

```
    Ok(res)
}

// If the contract is stored we make it executable and store the new state
pub fn try_activate(deps: DepsMut, contract_addr: Addr) -> Result<Response,
ContractError> {
    // this fails if contract is not available
    let mut contract = contracts().load(deps.storage, &contract_addr)?;

    // update the contract to be executable
    contract.is_executable = true;

    // and save
    contracts().save(deps.storage, &contract_addr, &contract)?;

    let res = Response::new().add_attributes(vec![
        ("action", "activate"),
        ("addr", contract_addr.as_str()),
    ]);
    Ok(res)
}

// If the contract has been stored, we make it not executable.
pub fn try_deactivate(deps: DepsMut, contract_addr: Addr) -> Result<Response,
ContractError> {
    // this fails if contract is not available
    let mut contract = contracts().load(deps.storage, &contract_addr)?;

    contract.is_executable = false;

    // and save
    contracts().save(deps.storage, &contract_addr, &contract)?;

    let res = Response::new().add_attributes(vec![
        ("action", "deactivate"),
        ("addr", contract_addr.as_str()),
    ]);
    Ok(res)
}

#[cfg_attr(not(feature = "library"), entry_point)]
pub fn query(deps: Deps, _env: Env, msg: QueryMsg) -> StdResult<Binary> {
    match msg {
        QueryMsg::GetContract { contract_address } => {
            to_binary(&query_contract(deps, contract_address)?)
        }
        QueryMsg::GetContracts { start_after, limit } => {
            to_binary(&query_contracts(deps, start_after, limit)?)
```

```
        }
        QueryMsg::GetActiveContracts { start_after, limit } => {
            to_binary(&query_active_contracts(deps, start_after, limit)?)
        }
    }
}

// Returns the ContractExecutionParams for contract_address
pub fn query_contract(deps: Deps, contract_address: Addr) ->
StdResult<ContractResponse> {
    let contract = contracts()
        .may_load(deps.storage, &contract_address)?
        .unwrap();

    let contract_info = ContractExecutionParams {
        address: contract_address,
        gas_limit: contract.gas_limit,
        gas_price: contract.gas_price,
        is_executable: contract.is_executable,
    };

    Ok(ContractResponse {
        contract: contract_info,
    })
}

// settings for pagination
const MAX_LIMIT: u32 = 30;
const DEFAULT_LIMIT: u32 = 10;

// Get all the contracts up to a limit, starting from start_address.
fn query_contracts(
    deps: Deps,
    start_after: Option<String>,
    limit: Option<u32>,
) -> StdResult<ContractsResponse> {
    let limit = limit.unwrap_or(DEFAULT_LIMIT).min(MAX_LIMIT) as usize;
    let addr = maybe_addr(deps.api, start_after)?;
    let start = addr.as_ref().map(Bound::exclusive);
    // iterate over them all

    // For each entry from state we generate a ContractExecutionParams
    let contracts = contracts()
        .range(deps.storage, start, None, Order::Ascending)
        .take(limit)
        .map(|item| {
            item.map(|(addr, contract)| ContractExecutionParams {
                address: addr,
```

```
                gas_limit: contract.gas_limit,
                gas_price: contract.gas_price,
                is_executable: contract.is_executable,
            })
        })
        .collect::<StdResult<_>>()?;
    Ok(ContractsResponse { contracts })
}

// Get all the contracts up to a limit, starting from start_address, that are active
(is_executable == true)
fn query_active_contracts(
    deps: Deps,
    start_after: Option<String>,
    limit: Option<u32>,
) -> StdResult<ContractsResponse> {
    let limit = limit.unwrap_or(DEFAULT_LIMIT).min(MAX_LIMIT) as usize;
    let addr = maybe_addr(deps.api, start_after)?;
    let start = addr.map(Bound::exclusive);
    // iterate over all and return only executable contracts
    let contracts = contracts()
        .idx
        .active
        .prefix(ACTIVE_CONTRACT)
        .range(deps.storage, start, None, Order::Ascending)
        .take(limit)
        .map(|item| {
            item.map(|(addr, contract)| ContractExecutionParams {
                address: addr,
                gas_limit: contract.gas_limit,
                gas_price: contract.gas_price,
                is_executable: contract.is_executable,
            })
        })
        .collect::<StdResult<_>>()?;
    Ok(ContractsResponse { contracts })
}
```

GENERAL QUESTIONS

1. **What are the concepts (borrowing, ownership, vectors etc)**
   The main concept at this point on the WBA is an implementation of an Indexed Map.

2. **What is the organization?**
   CosmWasm Smart Contract.

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

   Register, deregister, update, activate and deactivate contracts, which are indexed by their active attribute.

4. **How could it be better? More efficient? Safer?**
   Nothing that I can think of.