SMART CONTRACT	https://github.com/blasmorkai/cw-plus-bm/tree/main/contracts/cw20-base

Scope: query and execute entry points. (contract.rs.)

CODE REVIEW

Contract.rs

```
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn execute(
   deps: DepsMut,
   env: Env,
   info: MessageInfo,
   msg: ExecuteMsg,
) -> Result<Response, ContractError> {
   match msg {
       ExecuteMsg::Transfer { recipient, amount } => {
           execute_transfer(deps, env, info, recipient, amount)
       }
       ExecuteMsg::Burn { amount } => execute_burn(deps, env, info, amount),
       // Send is a base message to transfer tokens to a contract and trigger an
action
       // on the receiving contract.
       ExecuteMsg::Send {
           contract,
           amount,
           msg,
       } => execute_send(deps, env, info, contract, amount, msg),
       ExecuteMsg::Mint { recipient, amount } => execute_mint(deps, env, info,
recipient, amount),
       ExecuteMsg::IncreaseAllowance {
           spender,
           amount,
           expires,
       } => execute_increase_allowance(deps, env, info, spender, amount, expires),
       ExecuteMsg::DecreaseAllowance {
           spender,
           amount,
           expires,
       } => execute_decrease_allowance(deps, env, info, spender, amount, expires),
       // ExecuteMsg::TransferFrom - Only with "approval" extension.
       ExecuteMsg::TransferFrom {
           owner,
           recipient,
```

```rust
            amount,
        } => execute_transfer_from(deps, env, info, owner, recipient, amount),
        // ExecuteMsg::BurnFrom - Only with "approval" extension.
        ExecuteMsg::BurnFrom { owner, amount } => execute_burn_from(deps, env, info,
owner, amount),
        // ExecuteMsg::SendFrom - Only with "approval" extension.
        ExecuteMsg::SendFrom {
            owner,
            contract,
            amount,
            msg,
        } => execute_send_from(deps, env, info, owner, contract, amount, msg),
        ExecuteMsg::UpdateMarketing {
            project,
            description,
            marketing,
        } => execute_update_marketing(deps, env, info, project, description,
marketing),
        ExecuteMsg::UploadLogo(logo) => execute_upload_logo(deps, env, info, logo),
        ExecuteMsg::UpdateMinter { new_minter } => {
            execute_update_minter(deps, env, info, new_minter)
        }
    }
}

pub fn execute_transfer(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    recipient: String,
    amount: Uint128,
) -> Result<Response, ContractError> {
    // The transfer needs to have some content
    if amount == Uint128::zero() {
        return Err(ContractError::InvalidZeroAmount {});
    }

    // The address of the recipient should be valid
    let rcpt_addr = deps.api.addr_validate(&recipient)?;

    // The balance of the sender on the ledger will get 'amount' deducted from its
balance
    // If there is no balance, the balance will default() to zero and when subtracting
an amount on a Uint will trigger Error
    BALANCES.update(
        deps.storage,
        &info.sender,
        |balance: Option<Uint128>| -> StdResult<_> {
            Ok(balance.unwrap_or_default().checked_sub(amount)?)
        },
    )?;

    // The balance of the receiver on the ledger will get 'amount' added to its
balance.
    // If that address has no balance, it will default() to zero and add the amount.

    // USE CASE 3: Options inside closure
    // When there is not value in the Option, .unwrap_of_default() let's us work with
that field either way.
    BALANCES.update(
        deps.storage,
        &rcpt_addr,
        |balance: Option<Uint128>| -> StdResult<_> { Ok(balance.unwrap_or_default() +
amount) },
    )?;

    // Response created with attributes describing the actions perfomed
    let res = Response::new()
        .add_attribute("action", "transfer")
        .add_attribute("from", info.sender)
        .add_attribute("to", recipient)
        .add_attribute("amount", amount);
    Ok(res)
}

pub fn execute_burn(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    amount: Uint128,
) -> Result<Response, ContractError> {
    // The amount to burn can not be zero
    if amount == Uint128::zero() {
        return Err(ContractError::InvalidZeroAmount {});
    }

    // lower balance
    // If that address has no balance, it will default() to zero and reduce the
amount, which will produce an error.
    // The closure deals with an Option on Maps.
    BALANCES.update(
        deps.storage,
        &info.sender,
        |balance: Option<Uint128>| -> StdResult<_> {
            Ok(balance.unwrap_or_default().checked_sub(amount)?)
        },
```

```rust
    )?;
    // reduce total_supply
    // If the reduction is greater that the total supply it will produce an error
because info.total_supply is an unsigned integer
    // The closure deals with the stored struct in Item.
    TOKEN_INFO.update(deps.storage, |mut info| -> StdResult<_> {
        info.total_supply = info.total_supply.checked_sub(amount)?;
        Ok(info)
    })?;

    // Response created with attributes describing the actions perfomed
    let res = Response::new()
        .add_attribute("action", "burn")
        .add_attribute("from", info.sender)
        .add_attribute("amount", amount);
    Ok(res)
}

pub fn execute_mint(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    recipient: String,
    amount: Uint128,
) -> Result<Response, ContractError> {
    // The amount to mint can not be zero
    if amount == Uint128::zero() {
        return Err(ContractError::InvalidZeroAmount {});
    }

    // If the token has not been set up, an error will be generated.
    // This scenario should not happen as the token is generated upon instantiation
    let mut config = TOKEN_INFO
        .may_load(deps.storage)?
        .ok_or(ContractError::Unauthorized {})?;

    // The minter has to be the sender of this message, otherwise error.
    // config.mint (Option<MinterData>) is turned with as_ref() (Option<&MinterData>)
    // Option<&MinterData> - > &MinterData The option is turned into a result with
.ok_or() unwrapping the result with ?
    // we can then access the value inside the option, i.e. minter

    // USE CASE 1: Options as a named fields in a struct
    if config
        .mint
        .as_ref()
        .ok_or(ContractError::Unauthorized {})?
        .minter
        != info.sender
    {
        return Err(ContractError::Unauthorized {});
    }

    // update supply and enforce cap
    config.total_supply += amount;
    if let Some(limit) = config.get_cap() {
        if config.total_supply > limit {
            return Err(ContractError::CannotExceedCap {});
        }
    }
    TOKEN_INFO.save(deps.storage, &config)?;

    // add amount to recipient balance
    // If the rcpt_addr balance does not exist, it will default() to zero and the
minted amount added to its balance
    let rcpt_addr = deps.api.addr_validate(&recipient)?;
    BALANCES.update(
        deps.storage,
        &rcpt_addr,
        |balance: Option<Uint128>| -> StdResult<_> { Ok(balance.unwrap_or_default() +
amount) },
    )?;

    let res = Response::new()
        .add_attribute("action", "mint")
        .add_attribute("to", recipient)
        .add_attribute("amount", amount);
    Ok(res)
}

pub fn execute_send(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    contract: String,
    amount: Uint128,
    msg: Binary,
) -> Result<Response, ContractError> {
    // To transfer tokens to a contract and trigger an action on the receiving
contract, based on the msg param.

    // The amount sent can not be 0
    if amount == Uint128::zero() {
        return Err(ContractError::InvalidZeroAmount {});
    }
    let rcpt_addr = deps.api.addr_validate(&contract)?;

    // move the tokens to the contract
    // The tokens are subtracted from the senders balance
    BALANCES.update(
        deps.storage,
        &info.sender,
        |balance: Option<Uint128>| -> StdResult<_> {
            Ok(balance.unwrap_or_default().checked_sub(amount)?)
        },
    )?;

    // The tokens are registered/added under the contract address
    BALANCES.update(
        deps.storage,
        &rcpt_addr,
        |balance: Option<Uint128>| -> StdResult<_> { Ok(balance.unwrap_or_default() +
amount) },
    )?;

    // The contract will receive a message, including sender, amount and ...
    //the parameter msg included in ExecuteMsg::Send {contract, amount, msg,}
    let res = Response::new()
        .add_attribute("action", "send")
        .add_attribute("from", &info.sender)
        .add_attribute("to", &contract)
        .add_attribute("amount", amount)
        .add_message(
            Cw20ReceiveMsg {
                sender: info.sender.into(),
                amount,
                msg,
            }
            .into_cosmos_msg(contract)?,
        );
    Ok(res)
}

pub fn execute_update_minter(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    new_minter: Option<String>,
) -> Result<Response, ContractError> {
    // Load the Token_Info. If it has not been set up (should have been upon
instantiation), raise ContractError
    let mut config = TOKEN_INFO
        .may_load(deps.storage)?
        .ok_or(ContractError::Unauthorized {})?;

    // To update the minter the sender must be the minter. We extract the mint from
the optional mint
    let mint = config.mint.as_ref().ok_or(ContractError::Unauthorized {})?;
    if mint.minter != info.sender {
        return Err(ContractError::Unauthorized {});
    }

    // Beatiful to see how the Option<String> is turned into an Option<MinterData>
    // First the address is checked to be correct, then Option<Result> is transposed
to Result<Option>, extracting the Option
    // finally we map the address(option) to a MinterData struct(option).
    let minter_data = new_minter
        .map(|new_minter| deps.api.addr_validate(&new_minter))
        .transpose()?
        .map(|minter| MinterData {
            minter,
            cap: mint.cap,
        });

    config.mint = minter_data;

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

    //the value of the attribute extracts the minter address as a string if it exists.
    Ok(Response::default()
        .add_attribute("action", "update_minter")
        .add_attribute(
            "new_minter",
            config
                .mint
                .map(|m| m.minter.into_string())
                .unwrap_or_else(|| "None".to_string()),
        ))
}
```

```
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn query(deps: Deps, _env: Env, msg: QueryMsg) -> StdResult<Binary> {
    match msg {
        QueryMsg::Balance { address } => to_binary(&query_balance(deps, address)?),
        QueryMsg::TokenInfo {} => to_binary(&query_token_info(deps)?),
        QueryMsg::Minter {} => to_binary(&query_minter(deps)?),
        QueryMsg::Allowance { owner, spender } => {
            to_binary(&query_allowance(deps, owner, spender)?)
        }
        QueryMsg::AllAllowances {
            owner,
```

```rust
            start_after,
            limit,
        } => to_binary(&query_owner_allowances(deps, owner, start_after, limit)?),
        QueryMsg::AllSpenderAllowances {
            spender,
            start_after,
            limit,
        } => to_binary(&query_spender_allowances(
            deps,
            spender,
            start_after,
            limit,
        )?),
        QueryMsg::AllAccounts { start_after, limit } => {
            to_binary(&query_all_accounts(deps, start_after, limit)?)
        }
        QueryMsg::MarketingInfo {} => to_binary(&query_marketing_info(deps)?),
        QueryMsg::DownloadLogo {} => to_binary(&query_download_logo(deps)?),
    }
}

pub fn query_balance(deps: Deps, address: String) -> StdResult<BalanceResponse> {
    // The address queried is validated
    let address = deps.api.addr_validate(&address)?;
    // If there is no balance the value returned will be zero, thanks to
.unwrap_or_DEFAULT()
    let balance = BALANCES
        .may_load(deps.storage, &address)?
        .unwrap_or_default();
    Ok(BalanceResponse { balance })
}

pub fn query_token_info(deps: Deps) -> StdResult<TokenInfoResponse> {
    // We load the token info
    let info = TOKEN_INFO.load(deps.storage)?;
    // The TokenInfoResponse is the TokenInfo without the mint field.
    let res = TokenInfoResponse {
        name: info.name,
        symbol: info.symbol,
        decimals: info.decimals,
        total_supply: info.total_supply,
    };
    Ok(res)
}

pub fn query_minter(deps: Deps) -> StdResult<Option<MinterResponse>> {
    // Using the let var : Option<_> = match option_var { Some() => Some()}, None =>
None,
    // we return the option stored on TOKEN_INFO
    let meta = TOKEN_INFO.load(deps.storage)?;
    let minter = match meta.mint {
        Some(m) => Some(MinterResponse {
            minter: m.minter.into(),
            cap: m.cap,
        }),
        None => None,
    };
    Ok(minter)
}
```

GENERAL QUESTIONS

1. **What are the concepts (borrowing, ownership, vectors etc)**
   The Concepts in the code are Structs, Item, Map, Vectors, Tupples, Ownerships, Enum (Options, Results), Coins, mocking, tests, use of modules.
   The way this code handles and converts Options is very efficient. In one line of code, Options<T> are turned into Options<Q>, mapping and validating in the same line.

2. **What is the organization?**
   Entry points and the functions taking care of its messages.

3. **What is the contract doing? What is the mechanism?**
   The code reviewed implements the following functionality: transfer, burn, mint, send, update minter, query balance, query token info and query minter.

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

   Nothing that I can think of.