Documentation around design decisions -- Martino Agostini

(martino.agostini@gmail.com)

As I came up with ideas for dapps, I made some design decisions about structuring and organizing this project. Considering that the dapp has both front-end assets and back-end business logic, the project will likely consist of at least two parties, with one part for managing user interface components and another for the backend services the application provides.

Project Outline: Creation of my own currency called "DO" leverage an ERC20 smart contract with the possibility to send and receive through Metamask; the balance situation includes a list of transactions.

The solution: The front-end build provides an effective balance scorecard with the possibility to check and have in unique snapshot the amount you have sent/ remaining in your portfolio and list of transactions. For security reasons, when you log out, the list of previous transactions will be erased.

Segregating actors from types and utilities

In planning the architecture for the project, one common practice is to place the code for the main actor in one file with separate additional files for defining the types your program uses and utility functions that don't require an actor.

We set up the back-end logic for your dapp to consist of the following files:

a) with the functions that require an actor to send query and update calls.

b) with helper functions that can be imported for the actor to use.

c) with all of the data type definitions for your dapp

It would help if you considered how best to use query calls instead of functions that can perform queries or updates in the planning and design phase. That is a good general rule to follow and can be applied broadly to most categories of dapps. However, we need to consider the security and performance trade-off that queries don't go through consensus and do not appear on the blockchain. Considering that the dapp retrieves sensitive information, we leverage a Metamask to provide a unique source.

We leveraged the Ethereum platform as requested, and we tested on the testnet. The transaction limits and gas fees may be a problem for your use case. Opportunity for bridge and leverage cross-chain ie Polkadot could be considered in the future

Testing and infrastructure

During the unit tests with integration tests and end-to-end tests that can exercise the dependencies between the web app, its back end, and a realistic simulation of the blockchain platform.

On-chain logic

the basic ownership and transfer mechanics provided by standards, We leverage Openzeppelin ERC20 library to develop our own currency “AGO”

Wallet-based authentication

When a user wants to log in, the back end delivers a random value (or nonce) to sign, and the user creates a signature with their private key using a wallet such as MetaMask. The back end validates that the correct account created the signature and then updates the nonce to prevent it from being reused.

Accounts and permissions

Not all user accounts are equal, and when you're designing your minting app, you'll need a way to restrict administrative actions to a subset of accounts.

On-chain, this can be accomplished using role-based access control, which lets you limit certain operations to accounts that have been tagged with a particular role. If you're using Ethereum or an EVM-compatible network, the OpenZeppelin AccessControl contract is a great place to get started exploring more about implementing accounts and permissions

Operator accounts

In addition to providing your users with well-permission accounts, you might want to perform certain operations on behalf of your users — for example, initiating bulk token transfers or other smart contract operations. Smart contract standards often provide methods for authorizing an account to act on behalf of the token owner. The future application will include ERC-721's setApprovalForAll grants an operator account the permission to transfer any token owned by the owner account.

Upgradability (future development)

Most successful web applications evolve over time, but smart contracts are immutable by default and difficult to change. Before deploying to Mainnet, consider making your contracts upgradeable using something akin to the OpenZeppelin Upgrades plugin. This adds a layer of indirection that allows you to push updates to a deployed contract without requiring all users to connect to a new contract address.

Additional Future development

Include details regarding your crypto tokens, including their value, how many you intend to use, and which crypto token platform you intend to issue them. Another critical point to include is how you intend to allow investors to redeem their tokens. Other sections should also include your terms and conditions or a link to your website where they can find them.