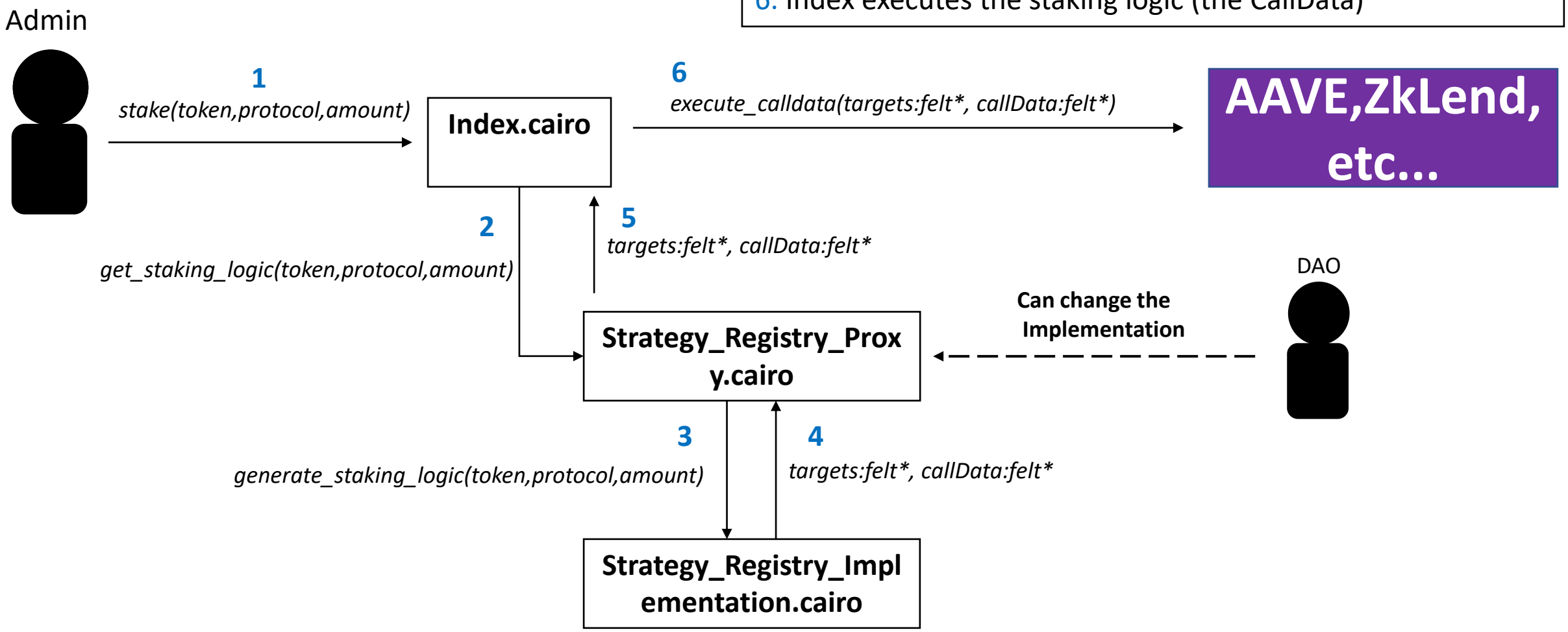
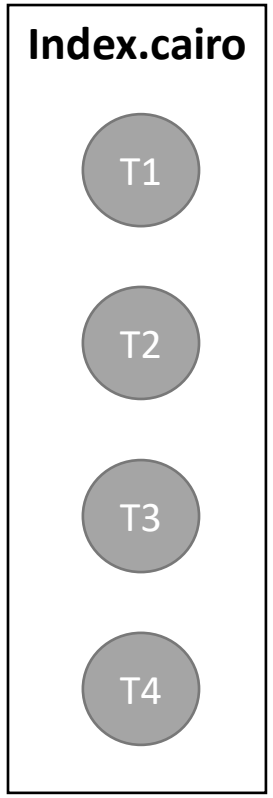
EXAMPLE IMPLEMENTATION

1. Admin makes a staking choice
2. Index requests the staking logic from the Registry Proxy
3. Proxy delegates the call to the implementation
4. Given, the token, amount and protocol, the Implementation generates the *felts* that can be used as CallData by the index
5. Proxy forwards CallData to Index
6. Index executes the staking logic (the CallData)

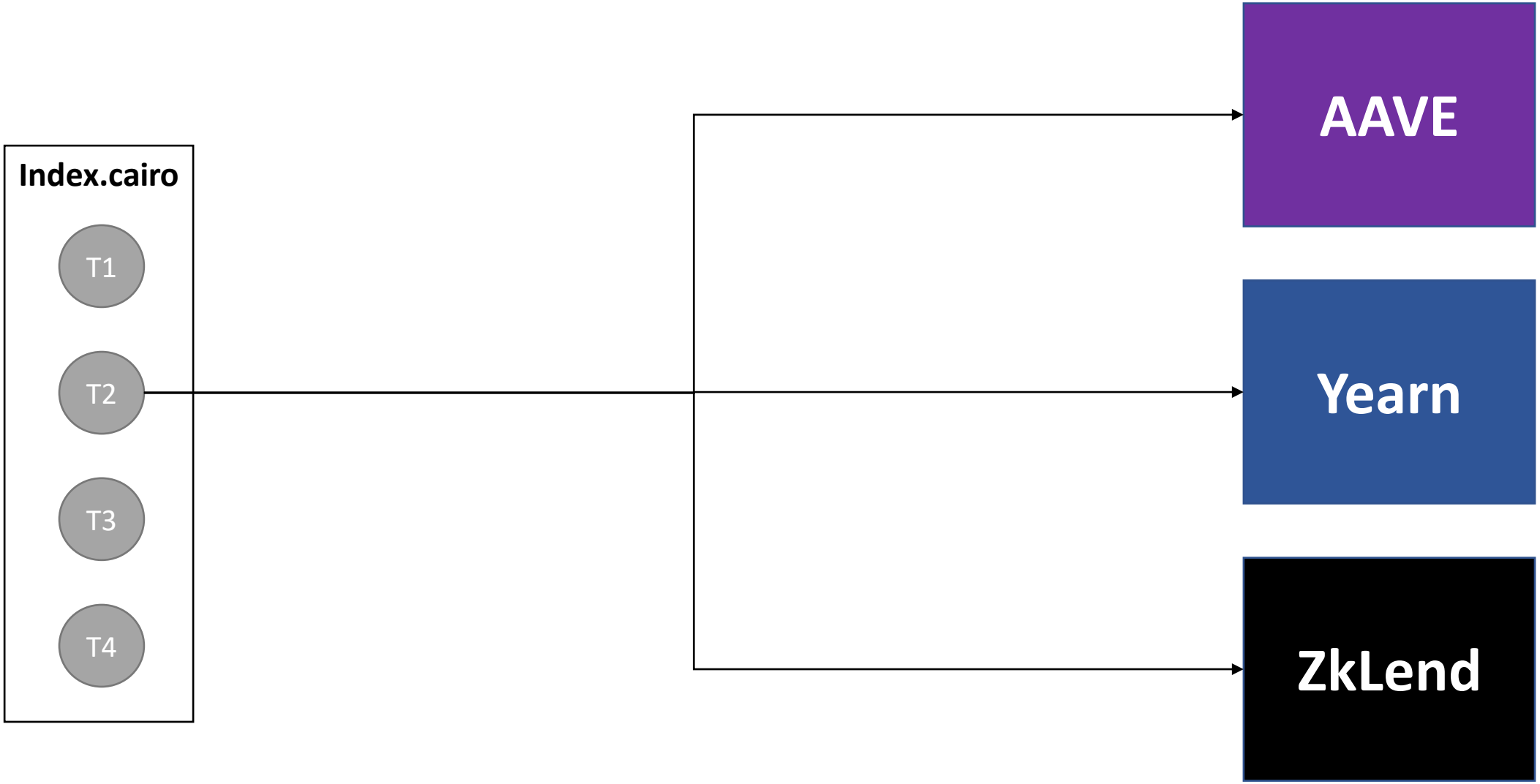


!!Draft!!

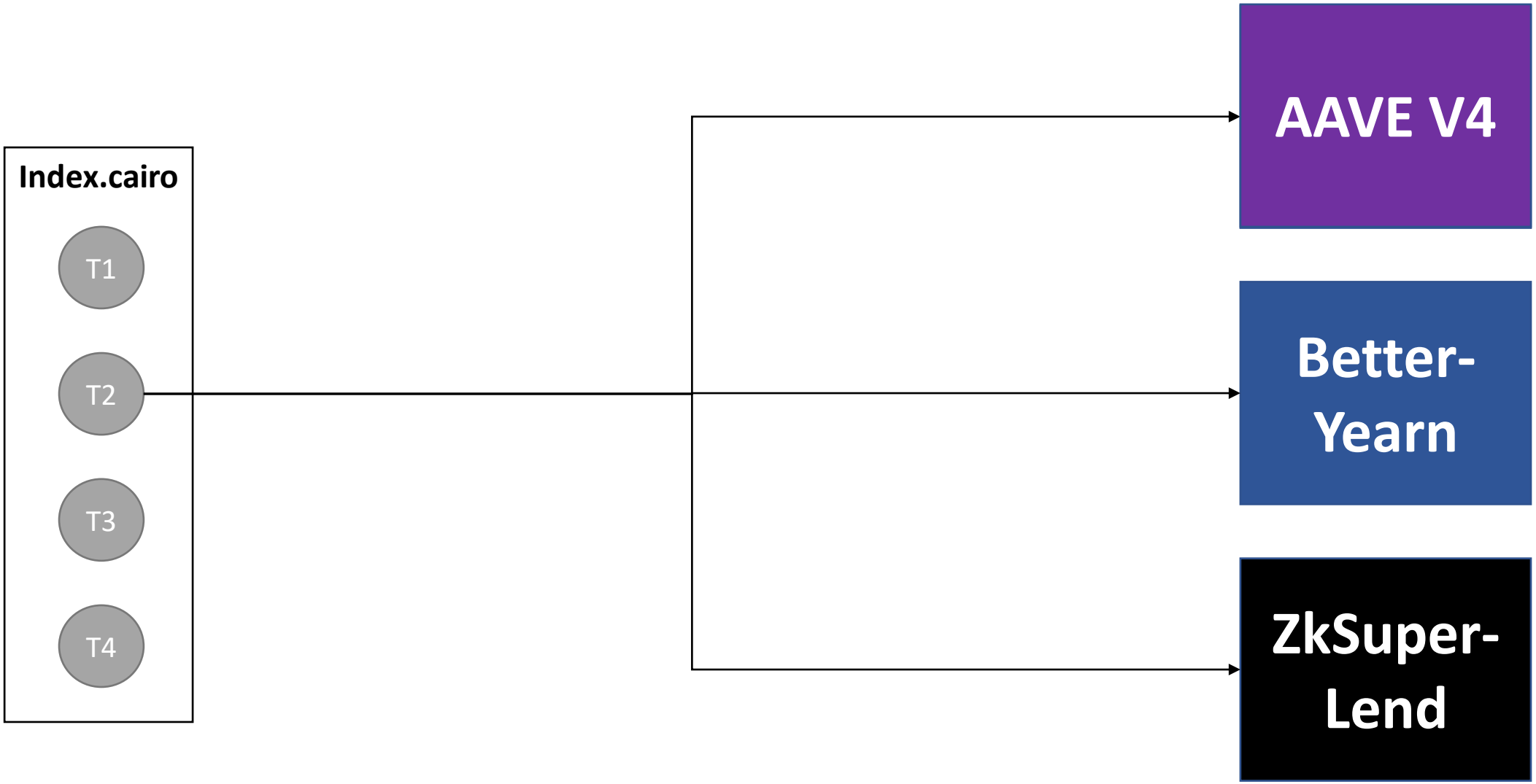
The following are some thoughts on what components are required for enabling yield bearing assets



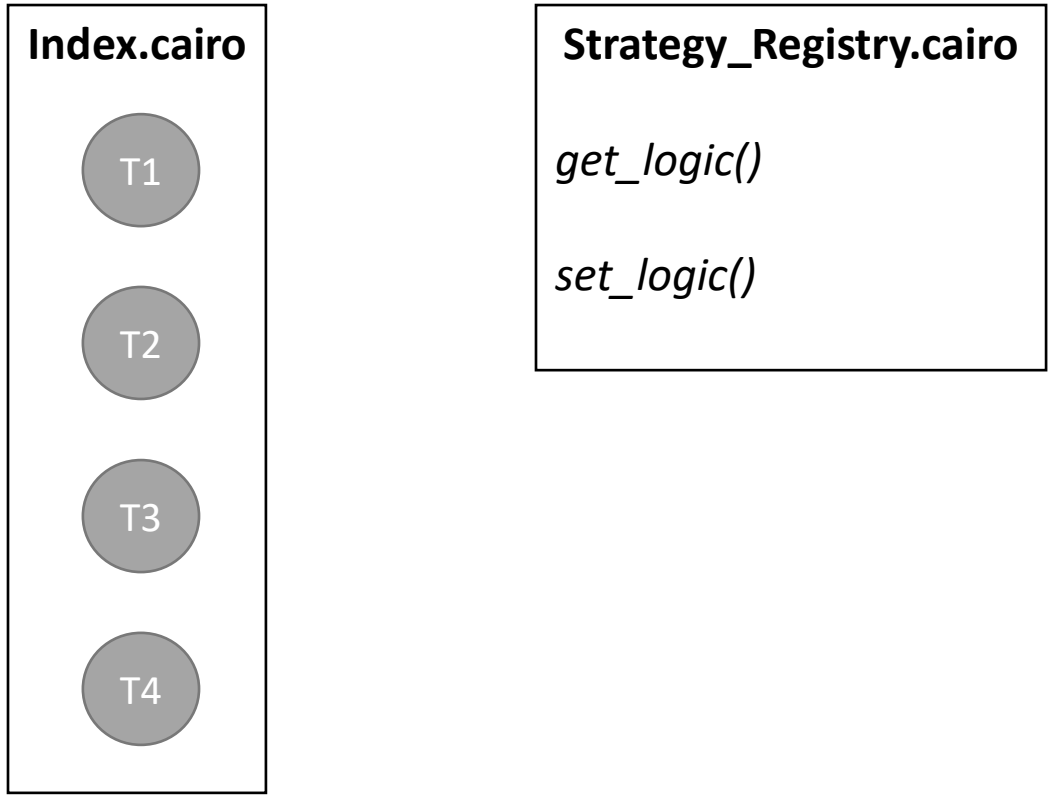
Although the standardization with ERC4626 will make it easier to write code that can be used for multiple protocols, custom code will still be needed regularly for those protocols that have implemented some unique staking logic.



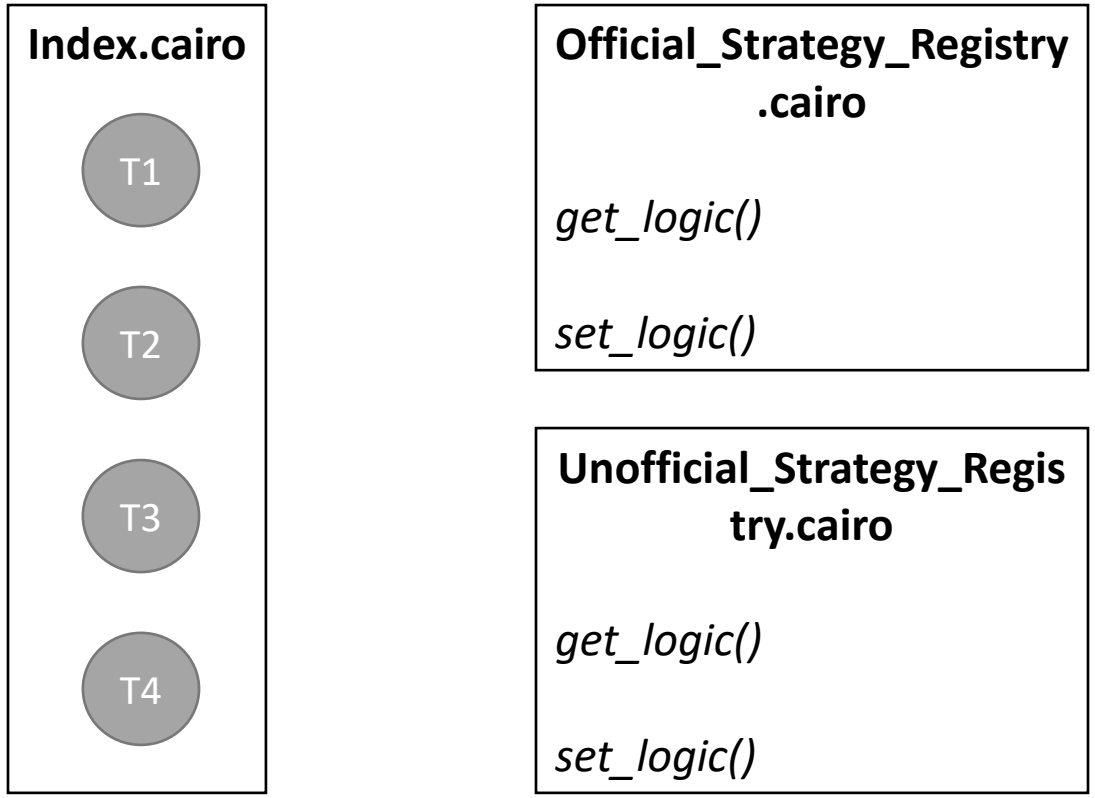
Furthermore, existing protocols may change, and with them the staking logic.



So, there is the need for a registry that holds the logic for staking assets in specific protocols. Existing logic can be changed or removed, and new logic can be added.

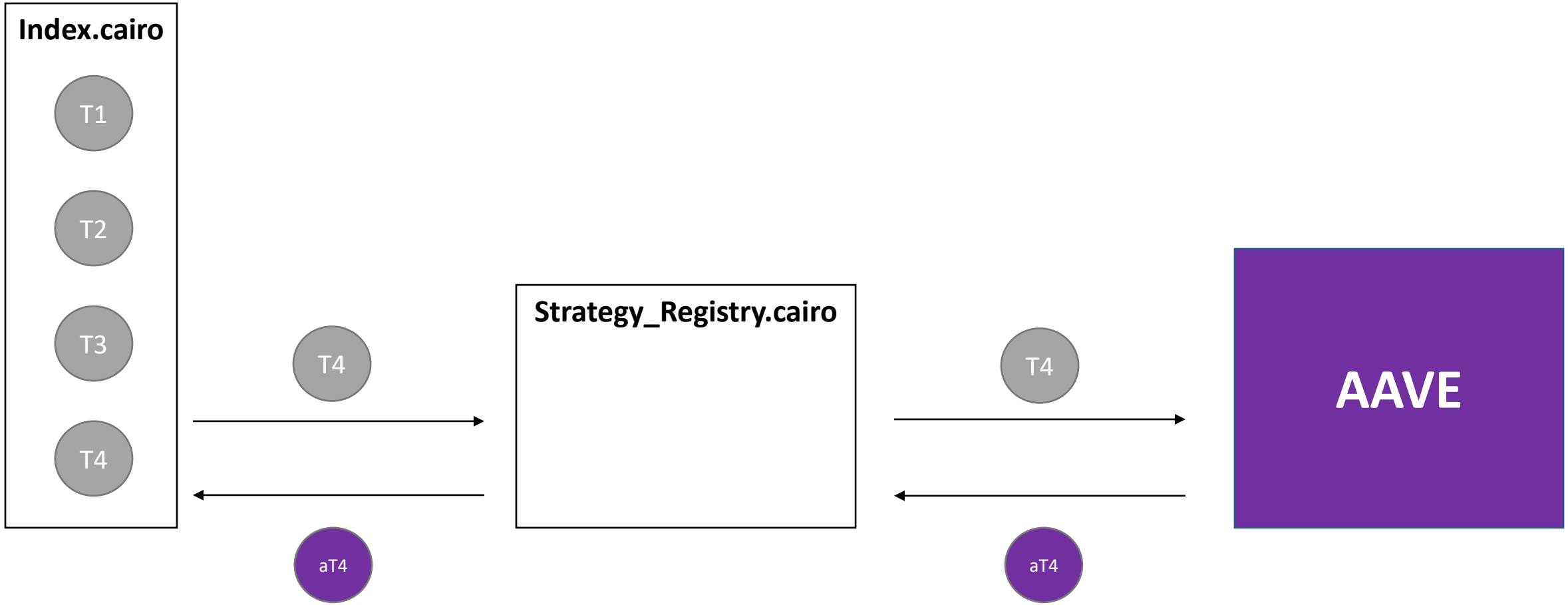


To ensure that the protocol is secure, but also as permissionless as possible, there should be a differentiation between **official** logic that was created by the DAO and logic that was added by a random entity.

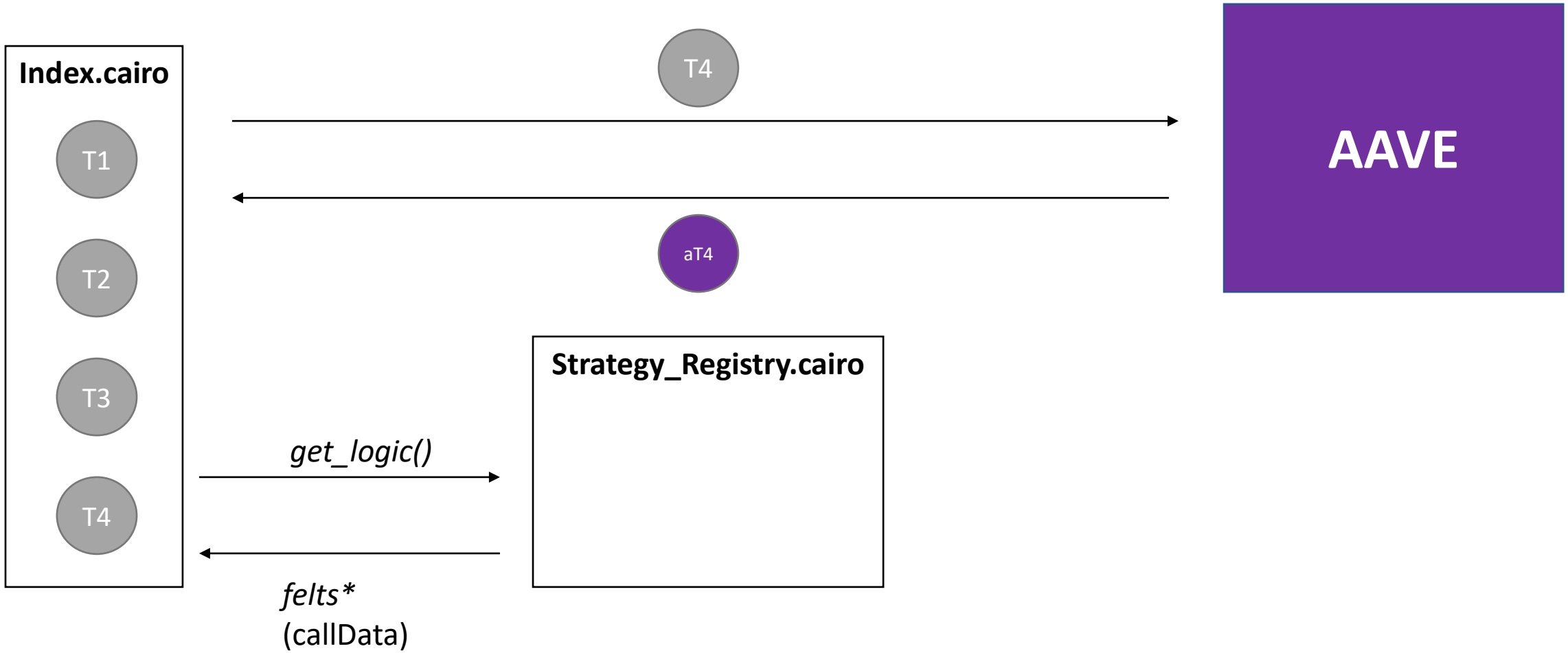


There are (at least) two options how the logic for staking the assets can be executed.

1.) The index just sends the underlying token to the registry, which then executes the staking logic and returns the wrapped token.



2.) The index fetches the callData from the registry and then executes the logic itself. (A similar result can be gained with delegateCall)



Option 1)
Cons: A lot of staking/lending protocols might have logic that is dependent on the msg.sender (so who ever the caller was). So e.g. rewards might be distributed to the registry and not the Index. The registry would require logic to handle these situations.

Option 2)
Cons: With option 1.) the exploitation risk is somewhat isolated to the registry, which doesn't hold any assets. Especially with any unofficial logic, there could be concern of the index interacting directly with malicious contracts. For the most part this can be negated by setting *approvals()* to the exact amounts needed and including re-entrancy protection.

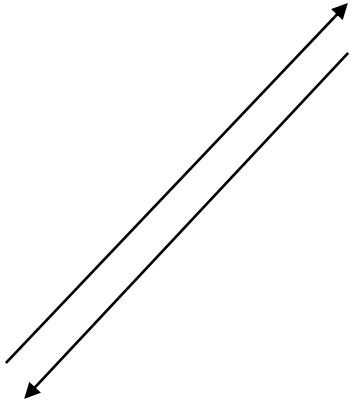
-Personally, I would choose Option 2

Strategy Registry Layout 1:
Separate Logic Contracts

Strategy_Registry.cairo

mapping (token,protocol,amount) -> (logic contract)

get_lendingLogic (token,protocol,amount) -> (contract_targets:felt*, callData:felts*)



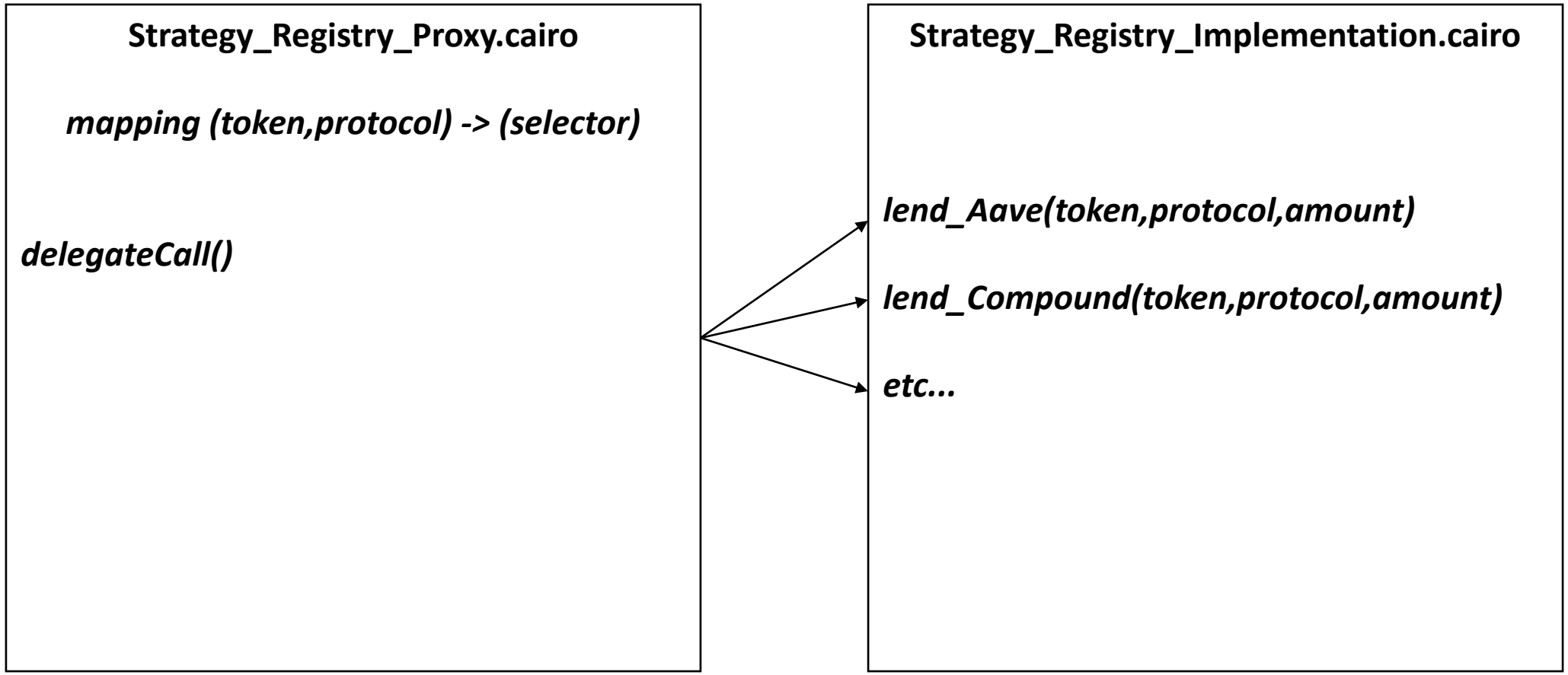
Lending_Logic_AAVE.cairo

get_lending_logic (token,protocol,amount) -> (contract_targets:felt*,callData:felts*)

Lending_Logic_Compound.cairo

Lending_Logic_ZkLend.cairo

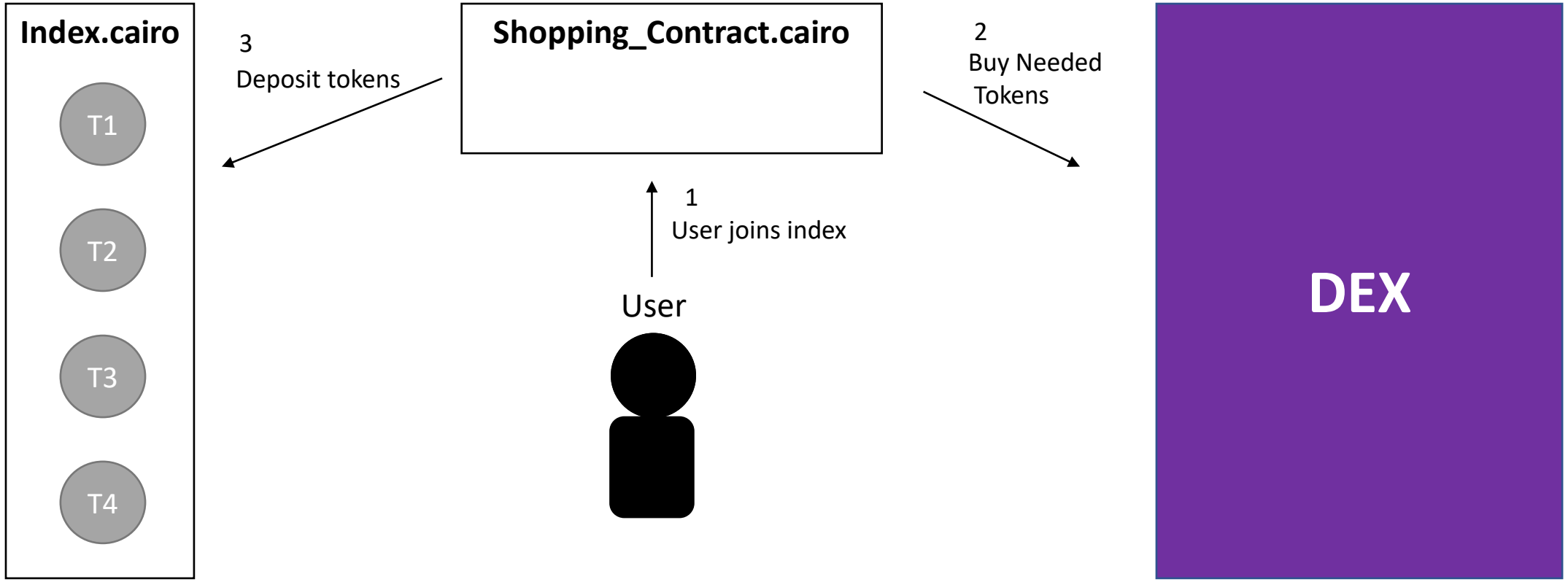
Strategy Registry Layout 2:
Single Contract for Staking Logic



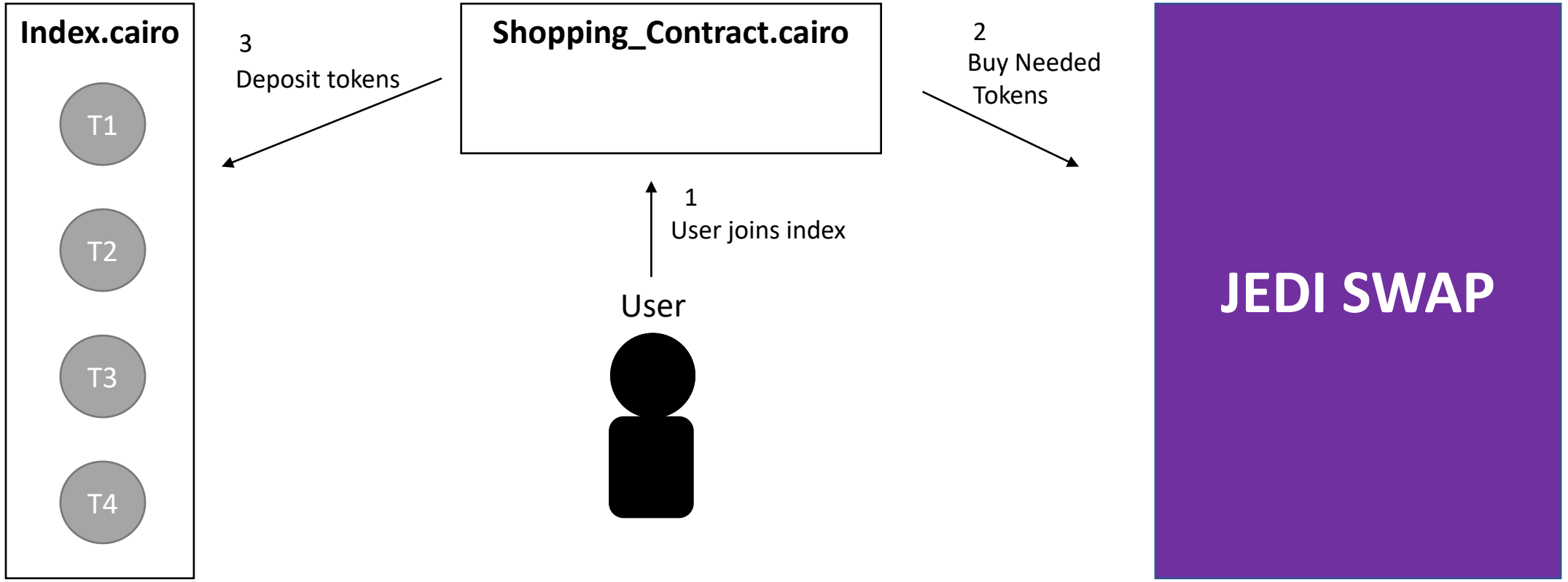
Who gets to execute the staking logic?

-Either the same role that gets to add/remove tokens
OR
-Separate Role only for staking/un-staking

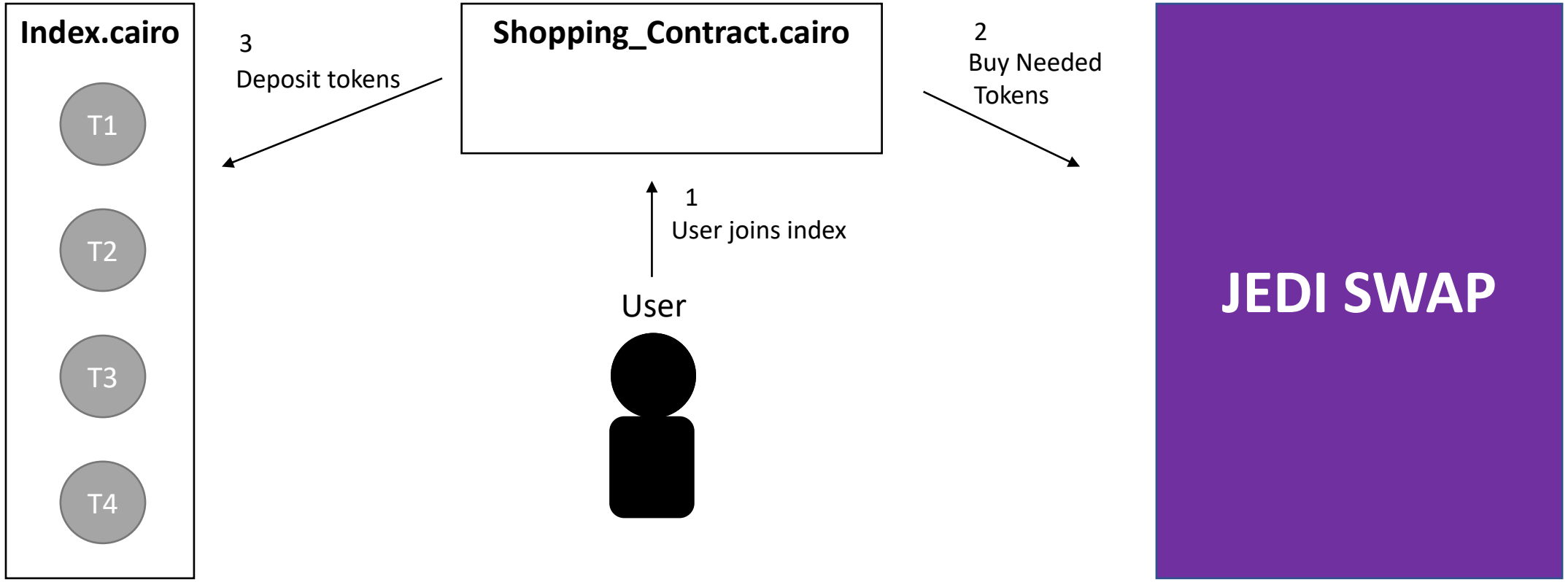
When a user wants to join the index, they either need to buy every asset in the index beforehand, or we require a contract that purchases everything for them and deposits it in the index with a single transaction.



This way we can also control where the assets are bought....wink wink



The issue here is, that the shopping_contract will see that the staked tokens are in the index, which cannot be bought on a DEX. So, it needs to know the conversion rate from the *staked-token* version to the *underlying-token*, in order to buy them.



The Strategy Registry contract should therefore also be forced to provide an exchange rate formular for every available staking protocol.
Realistically, it also requires a mapping for every *stakedtoken*, to query if it is in fact a *normal token* or a *staked token*.