THINGS WORTH NOTICING / THINGS TO CLARIFY

**APTOS MOVE**

**Worth noticing**

<https://aptos.dev/guides/creating-a-signed-transaction/>

* Workflow to create a signed transaction
* Structure of a transactions

<https://aptos.dev/guides/interacting-with-the-aptos-blockchain/>

* *While many blockchains implement a set of native operations, Aptos delegates all operations to Move, including: account creation, fund transfer and publishing Move modules*.

This reinforces the idea that an implementation of Move onto Algorand would compile Move operations on resources to Algorand transactions on corresponding assets.

* *Within a given transaction, the target of execution can be one of two types:*

*An entry point,*

*A script (payload)*

*Currently the SDKs*[*Python*](https://github.com/aptos-labs/aptos-core/blob/b0fe7ea6687e9c180ebdbac8d8eb984d11d7e4d4/ecosystem/python/sdk/aptos_sdk/client.py#L249)*and*[*Typescript*](https://github.com/aptos-labs/aptos-core/blob/76b654b54dcfc152de951a728cc1e3f9559d2729/ecosystem/typescript/sdk/src/aptos_client.test.ts#L98)*support the generation of transactions that target entry points only. This guide points out many of those entry points, such as coin::transfer and aptos\_account::create\_account.*

*All operations on the Aptos blockchain should be available via entry point calls. While one could submit multiple transactions calling entry points in series, many such operations may benefit from being called atomically from a single transaction. A script payload transaction can call any entry point or public function defined within any module.*

This confirms that function calls and transactions are orthogonal concepts.

**Needing a clarification**

<https://aptos.dev/guides/move-guides/move-on-aptos/>

* “*The complexity with generics arises when it would be desirable to store data on T. Move does not support static dispatch on generics, hence in a function like create<T>(...) : Coin<T>, T must either be a phantom type, i.e., only used as a type parameter in Coin or it must be specified as an input into create. No functions can be called on a T, such as T::function even if every T implements said function*.”. My understanding is as follows:
  1. unlike Rust, Move does not implement generics à la C++, as code stubs to be instantiated at each type at which they are used;
  2. that is good, but it would require the code to be polymorphic;
  3. that, in turn, is hardly possible given the following choice: “*structs can store any non-reference type, including other structs*” (cf here: <https://aptos.dev/guides/move-guides/book/structs-and-resources>): why is that, by the way?
  4. In fact, because of that choice, there is no way to instantiate a polymorphic struct, as we do not have a fixed size for its fields.

Mmm, the above explanation is unconvincing: one could easily store a pointer to the struct without explicitly representing the pointer (though having explicit references in the language …).

Also, see the discussion on generics: <https://aptos.dev/guides/move-guides/book/generics>

<https://aptos.dev/tutorials/your-first-transaction/>

* “*Step 4.5: Transferring: Like the previous step, this is another helper step that constructs a transaction transferring the coins from Alice to Bob. For correctly generated transactions, the API will return a transaction hash that can be used in the subsequent step to check on the transaction status. The Aptos blockchain does perform a handful of validation checks on submission; and if any of those fail, the user will instead be given an error. These validations use the transaction signature and unused sequence number, and submitting the transaction to the appropriate chain*.”

The code that illustrates this step seems to have a problem: the python\_sdk version appears to be calling the “transfer\_coins” function from “0x1::aptos\_account” rather than the “transfer” function from “0x1::coin::transfer”. The typescript version, instead, looks ok (cf the comment and the code below):   
 // If we should create the receiver account if it doesn't exist on-chain,  
 // use the `0x1::aptos\_account::transfer` function.  
 *const* func =

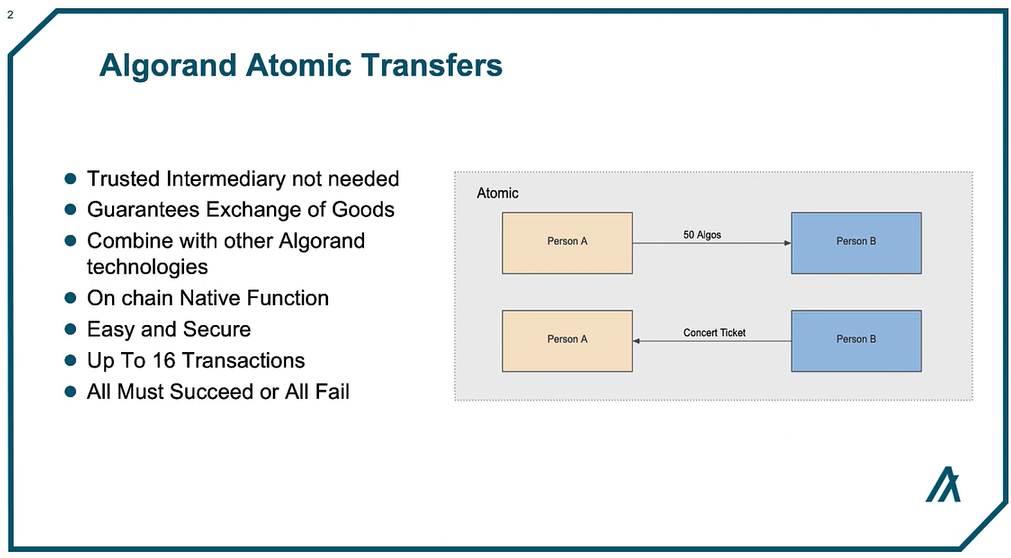
extraArgs?.createReceiverIfMissing ? "0x1::aptos\_account::transfer\_coins" : "0x1::coin::transfer";

**AVM / ALGORAND EXPLAINED**

Check here <https://www.youtube.com/@algodevs> for a set of useful video series. Some of them are commented below.

**ATOMIC TRANSACTIONS**

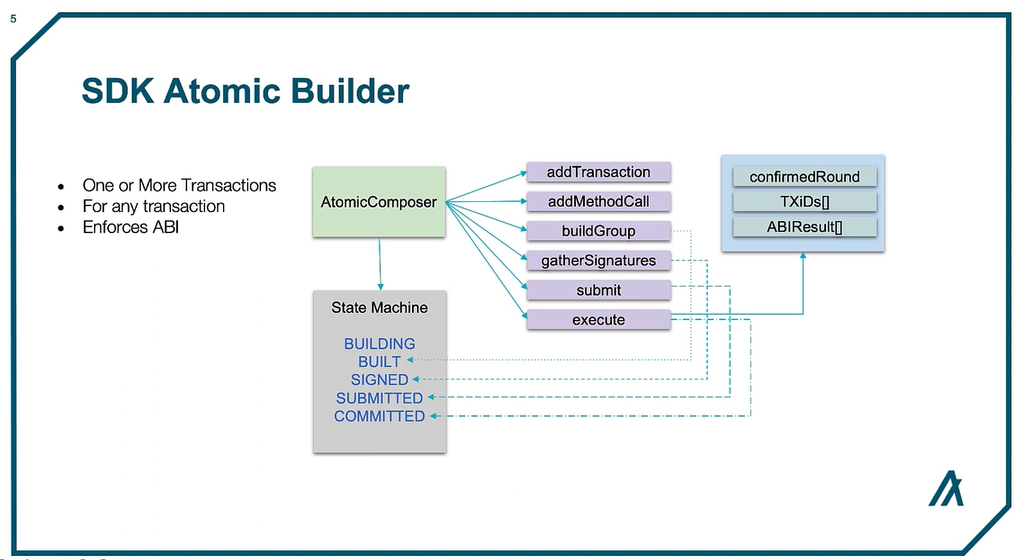
* <https://www.youtube.com/watch?v=65I0wyq7XoI&list=PLwRyHoehE437lw1IiQy_4AJVn95zm4YVp&index=1>

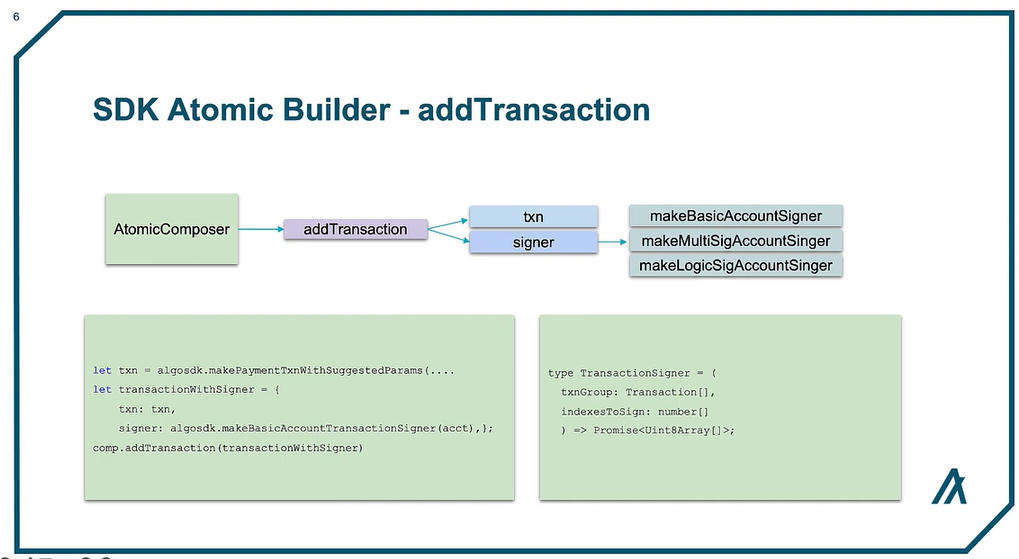


To be noticed:

* We are talking about groups of transactions as the atomic unit of execution.
* Transactions are native on-chain functions, i.e. they are native system calls. We have 16 of them, one of which – Application Call Transactions – activates smart contracts (see the section on Smart Contracts below), while the others can be thought of a language primitives (implemented as syscalls).

Below we can see how we build transaction groups:

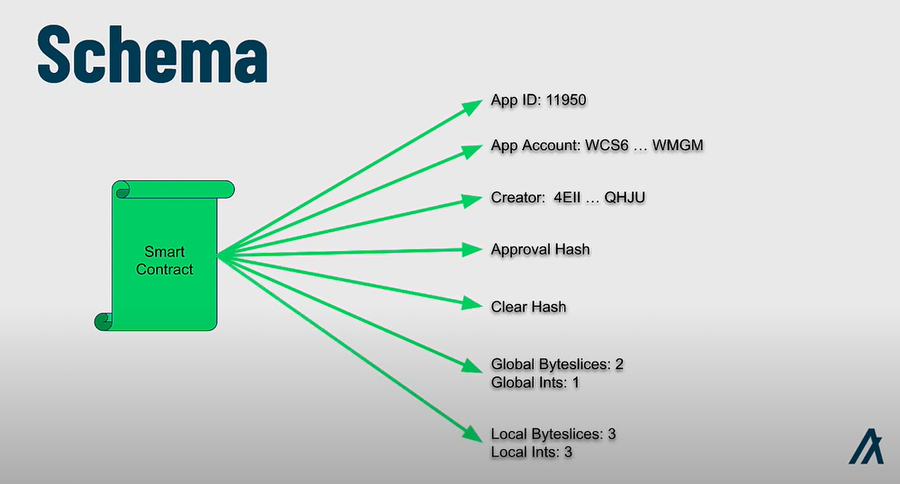




* <https://www.youtube.com/watch?v=65I0wyq7XoI&list=PLwRyHoehE437lw1IiQy_4AJVn95zm4YVp&index=2>

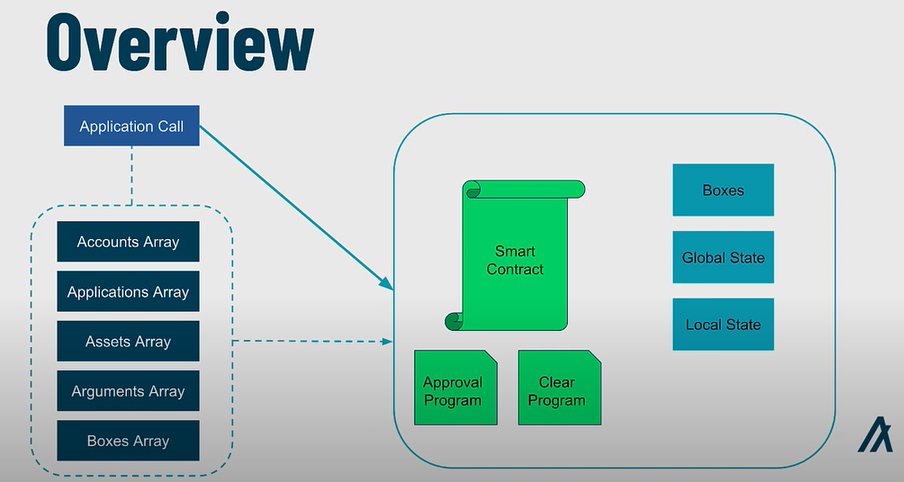
**SMART CONTRACTS**

* <https://www.youtube.com/watch?v=65I0wyq7XoI&list=PLwRyHoehE437lw1IiQy_4AJVn95zm4YVp&index=4>

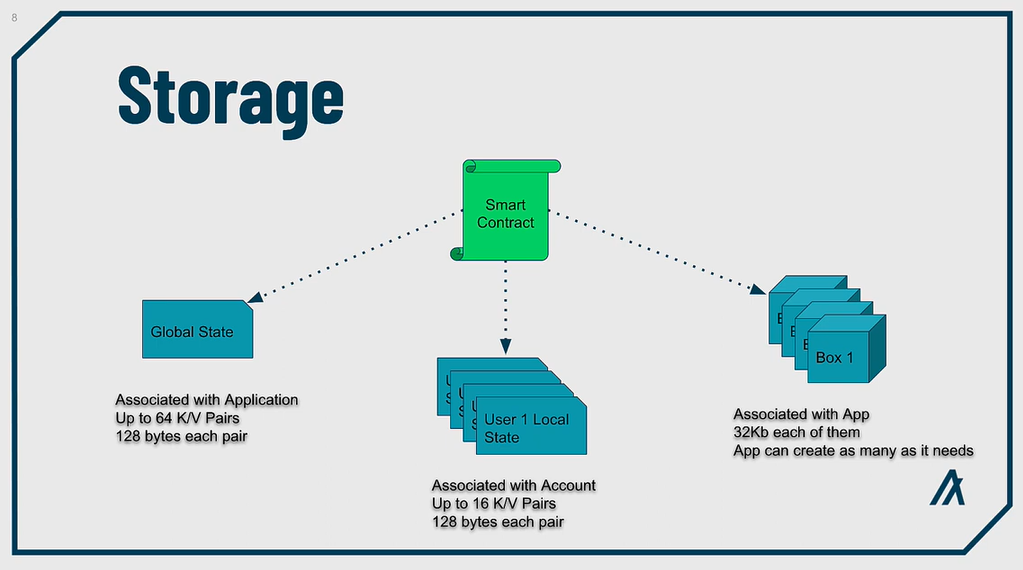


To be noticed:

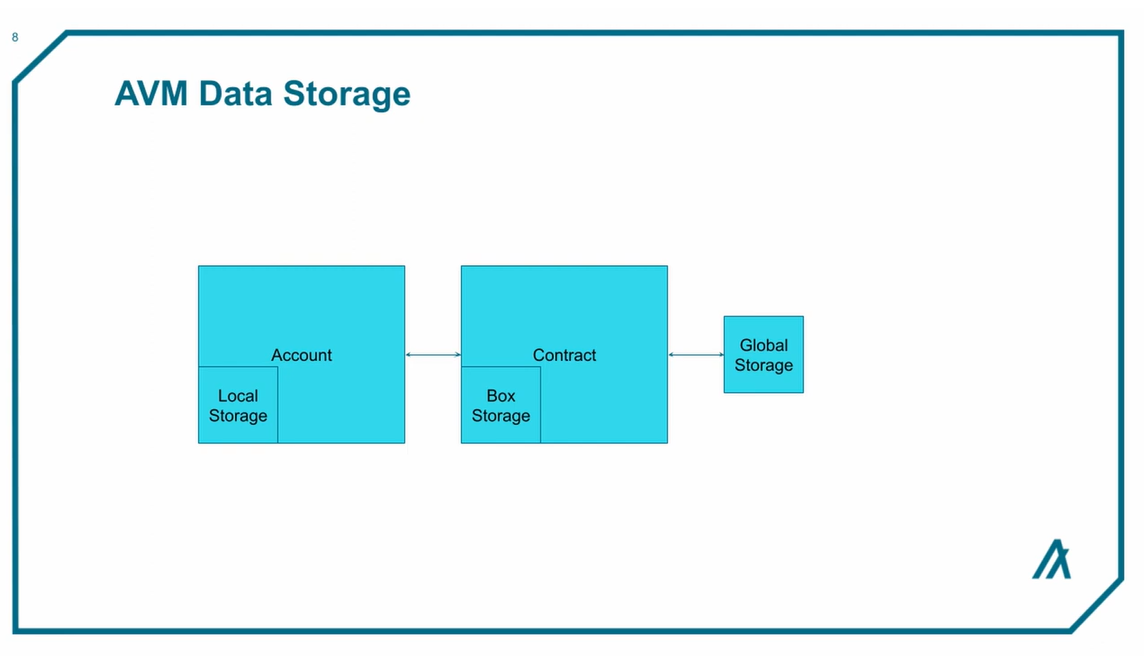
* Each smart contract has an associated account to deal with assets
* Below we see the general structure of the smart contract (a.k.a. application)

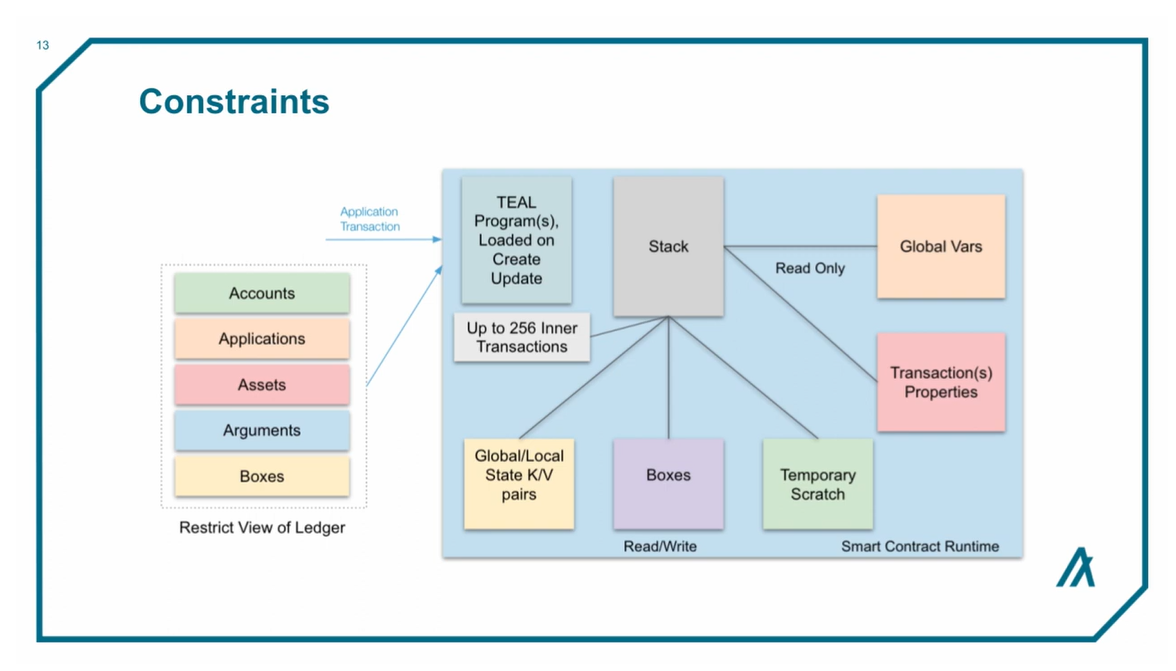


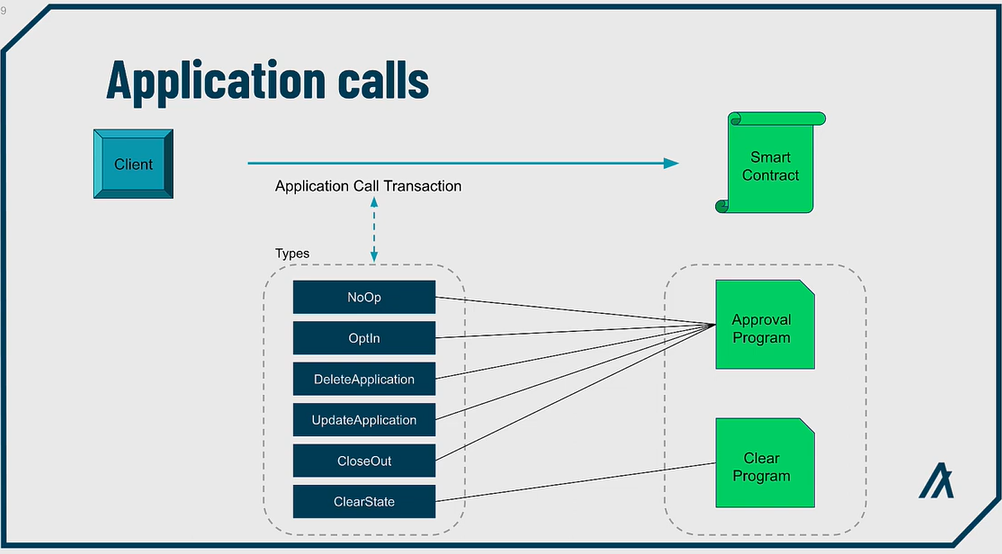
Here is the storage: global state and boxes are associated with the smart contract, while local storage is associated with the accounts that opt in the contract (i.e. that invoke the contract with an opt-in call, see the slide transactions below)



See also these two slides from the [AMV explained series](•%09https:/www.youtube.com/watch?v=96pwBo5jqnk&list=PLwRyHoehE437GKZsksMrc2RW932TC6mZJ&index=1)

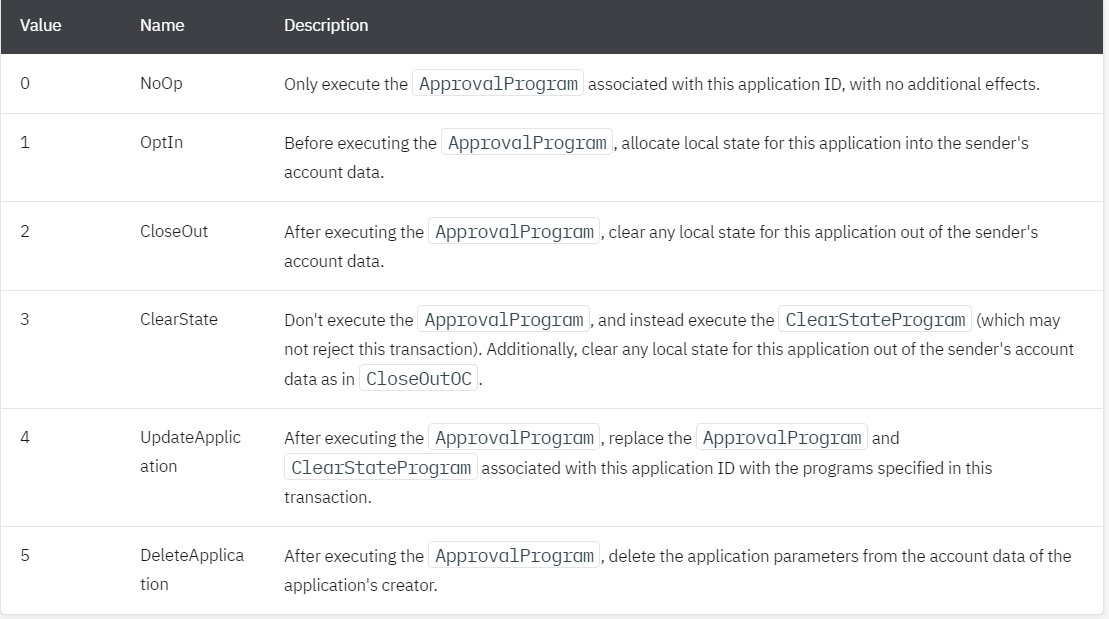




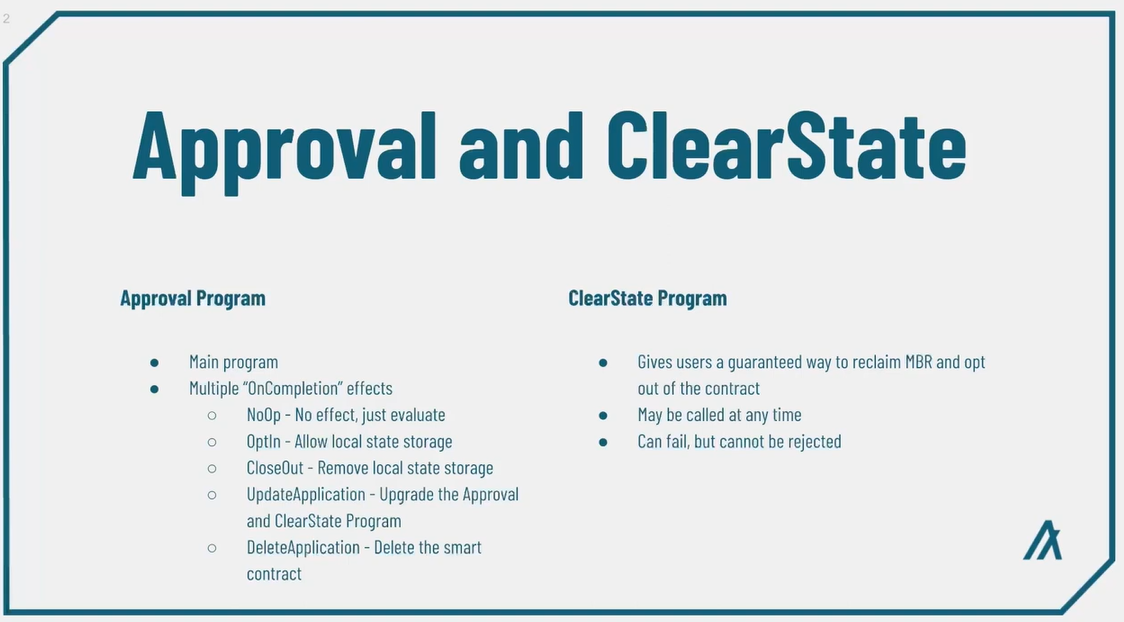


To be noticed:

* Application transactions are just one of the transactions types available (a detailed description is here: <https://developer.algorand.org/docs/get-details/transactions/>). As explained there, an Application Call Transaction is submitted to the network with an AppId and an OnComplete method. The AppId specifies which App to call and the OnComplete method is used in the contract to determine what branch of logic to execute, as shown above.
* The types of the Application Call Transactions correspond to the OnComplete “method” set in the transaction, as shown in the following table found at this link: <https://developer.algorand.org/docs/get-details/dapps/avm/teal/specification/#oncomplete>



* and similarly in the following slide from



* See here: <https://developer.algorand.org/docs/get-details/transactions/transactions/> for further details about transactions.

