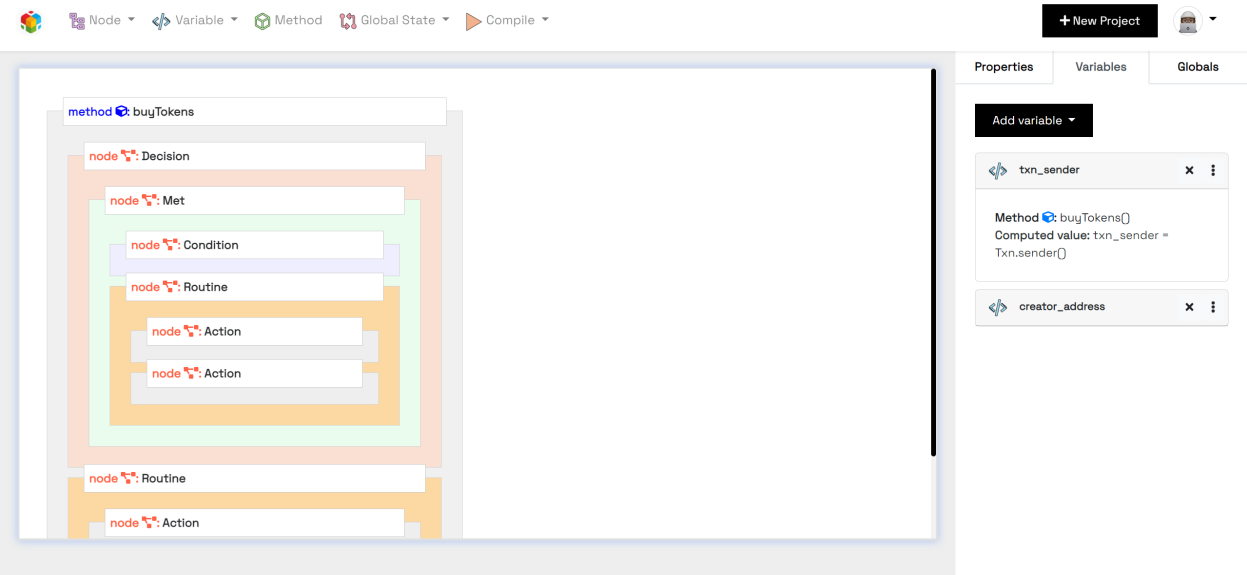
ASC Builder

AlgoKnox

Jul 02, 2022

USER GUIDE

ASC Builder is a drag-and-drop GUI tool that makes it possible for developers to create Pyteal source code that can in turn be compiled to TEAL smart contracts by the use of UI widgets, actions and configurations that specify their behaviours.



The aim of the project is to make the development of smart contracts easier and provide an open and shared workflow that features exports and imports of projects via JSON schemas.

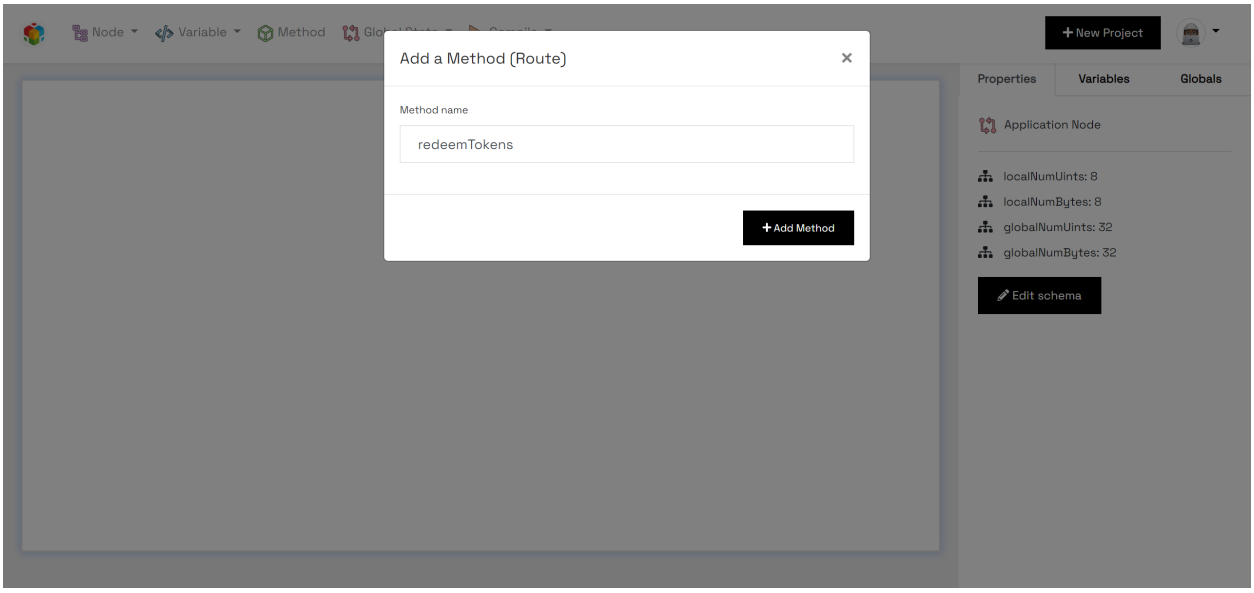
It also helps in preventing unforseen errors when constructing nodes, methods and actions.

CHAPTER

ONE

METHODS

A method (or route) can be added to the GUI by clicking on the 'Method' menu item at the top-right part of the screen.



You then have to provide the name of the method, which **MUST** be named with the variable naming standards as used in Python.

For instance; if a method for sending tokens to the caller of the smart contracts is to be created, you can use a name such as

redeemTokens or **redeem_tokens** or **RedeemTokens**

Method names must not begin with numbers!

Furthermore, a Method cannot be added to another Method.

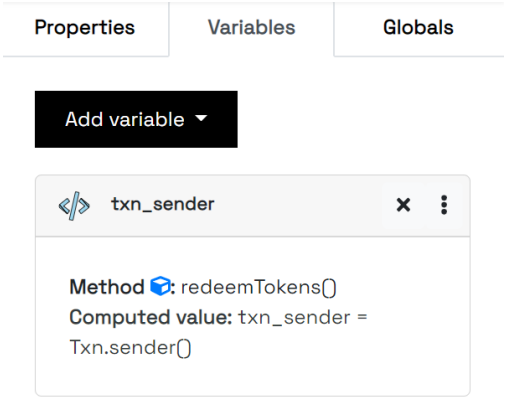
CHAPTER

TWO

VARIABLES

There are many variables that can be added to the GUI that help to hold different values that range from Transaction properties to Integers, Byte Slices and many more.

Variables that are created are added to the Variables side panel at the right-hand side of the screen. To create a variable, you specify the variable name and construct its value from the UI.



There are a total of *18* variables that can be created thus;

2.1 Address

An Address variable is meant to hold a valid Algorand Address. There are many options that can be chosen as values for the Address variable and are selected from a dropdown.

Name	Description
Address	This is used to specify a 58-character Algorand address that you intend to be stored in the variable. It's a string representation of a valid address.
Creator Address	This represents the Address of the creator of the smart contract. Equivalent to `Global.creator_address()` in Pyteal.
Current Application Address	This represents the Escrow account attributed to the smart contract. Equivalent to `Global.current_application_address()`.
Accounts	This is used to provide a positive integral index in an array of accounts specified in the Application Call. Equivalent to `Txn.accounts[]` in Pyteal.
Sender	This represents the caller (sender) of the transaction. Equivalent to `Txn.sender()`.
Receiver	This represents the receiver of a Payment transaction. Equivalent to `Txn.receiver()`.
Asset Receiver	This represents the receiver of an AssetTransfer transaction. Equivalent to `Txn.asset_receiver()`.

2.2 Asset

An Asset variable is meant to hold an asset index that can be used for an asset transfer or any other ASA operation. There are two means of specifying the value of this variable:

Name	Description
Asset Index	This is used to specify the ASA ID of the asset for any Asset operation.
Assets	This is used to specify an positive integral index in the assets provided in the Application Call.

2.3 Byte String

A Byte String variable is meant to provide a string variable in the application. It result to `Bytes()` in Pyteal.

2.4 Integer

An Integer variable is meant to provide a positibe numeric variable in the application. It results to `Int()` in Pyteal.

2.5 Asset Holding

An Asset Holding variable is meant to provide the balance of an asset ($ALGO or ASA) of a specified Algorand address. You also specify the account in question as a variable (most specifically an Address variable) whose balance should be gotten for the given asset.

Possible choices for the target asset are:

Name	Description
$ALGO	This is used to get the $ALGO balance of an asset in MicroAlgos.
Asset Index	This is used to specify the Integral value of the ASA in which to get the account's balance for.
Variable Name	This is used to specify a variable name that contains the ASA in question.

2.6 Min Balance

A Min Balance variable is meant to get the minimum possible balance of an Algorand account. The required parameter to provide is a variable that holds the Account whose minimum balance should be retrieved. Equivalent to `MinBalance()` in Pyteal.

2.7 Asset Decimals

An Asset Decimals variable is meant to get the decimals of an asset as an integral value such as *0*, *1*, *2* etc. The parameter provided is either an Asset Index or a variable that holds the ASA ID of the asset whose decimals you want to get. Equivalent to `AssetParams.decimals()` in Pyteal.

2.8 Argument

An Argument variable is used to get a 1-indexed value in the array of arguments provided in the Application Call. Equivalent to `Txn.application_args[]` in Pyteal.

2.9 Application

An Application variable is used to represent an application ID. Possible values that can be provided are a 1-indexed index in the list of applications (`Txn.applications[]`) provided in the Application Call or an integral value for the application ID (Note that it must still be in the list of applications).

2.10 Transaction Field

A Transaction Field variable is used to represent a transaction field such as sender, fee, receiver etc.

2.11 Grouped Transaction Field

A Grouped Transaction Field variable is used to represent a field in a grouped transaction. The group index and the field are values that should be provided for this variable.

2.12 Miscellaneous

A Miscellaneous variable is used to create variables that perform special transforms to variables such as arithmetic operations, ByteSlice manipulation and type casting.

2.13 Local State

A Local State variable is used to hold a local state key from a specified Algorand account. Equivalent to `App.localGet()`

2.14 Global State

A Global State variable is used to hold a global state value in the application. Equivalent to `App.globalGet()`

2.15 External Local State

An External Local State variable is used to hold a local state value for an Algorand account in an external application whose ID is provided. Equivalent to `App.localGetEx()`

2.16 External Global State

An External Global State variable is used to hold a global state value in an external application whose ID is provided. Equivalent to `App.globalGetEx()`

2.17 Global Field

A Global Field variable is used to hold a `Global` value such as the latest timestamp, creator address and many others.

2.18 Transaction Type

A Transaction Type variable is used to hold a value that represents a type for a transaction, which can be used for comparison to the `type_enum()` field of a transaction.

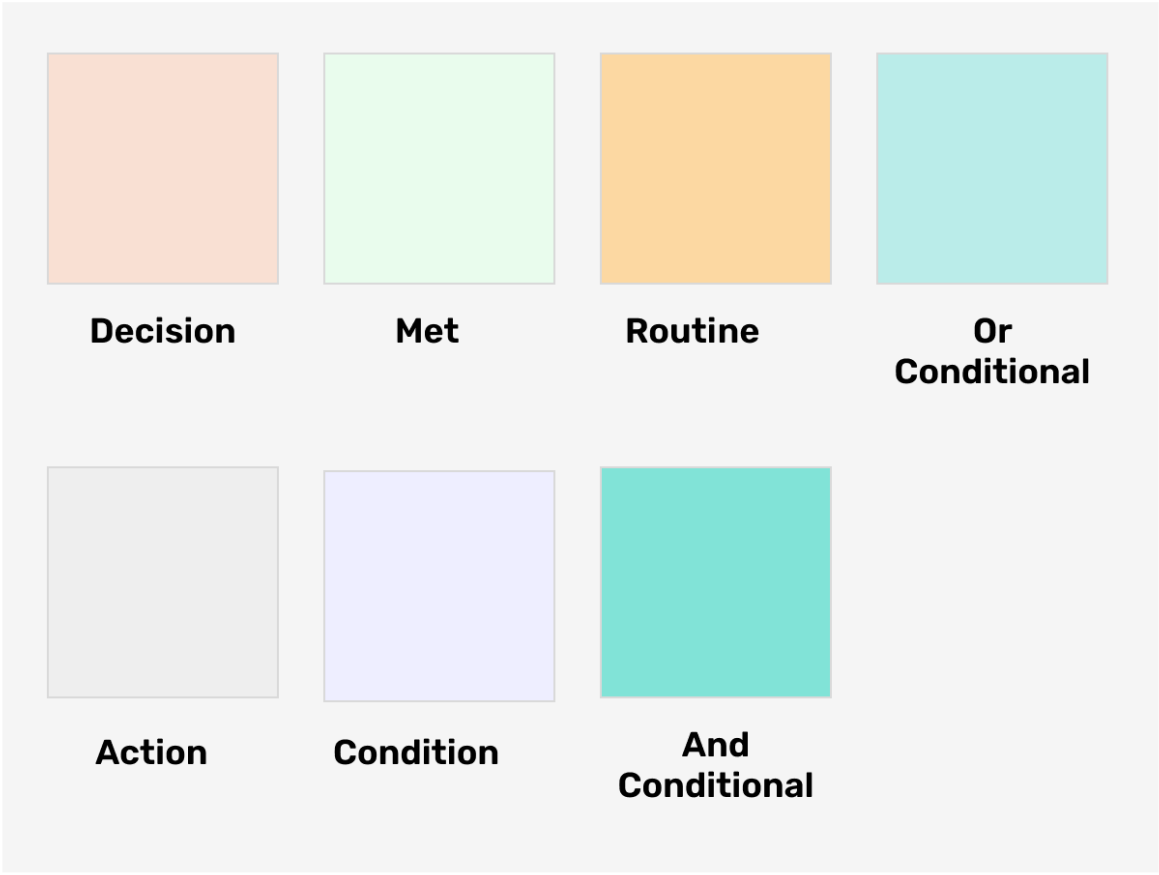
CHAPTER

THREE

NODES

A Node is a component that represents a flow in the GUI program. It can be used to define the structure and behaviour of a method in your smart contract.

Each Node has a special color attributed to it to enable easy differentiation and streamlining.



There are quite number of nodes that can be used in ASC Builder, which are as follows:

3.1 Decision

A Decision node is a wrapper for the `If` statement and helps to construct logical decisions. A typical decision Node consists of a Met node and a Routine node.

3.2 Met

A Met node is a used to specify a condition or set of conditions (And Conditional, Or Conditional) and an action or group of actions to be performed when met.

3.3 Routine

A Routine node is used to specify an action or set of actions to be performed in your smart contract.

3.4 Or Conditional

An Or Conditional node is used to specify a set of conditions in such a way that at least one of any of the conditions must be met for it to be valid.

3.5 And Conditional

An And Conditional node is used to specify a set of conditions in such a way that all of the conditions must be met for it to be valid.

3.6 Action

An Action node is used to perform an operation such as an asset transfer, approval, rejection, etc. in your smart contract. Some examples of actions are:

1. Send $ALGO
2. Send an ASA
3. Opt-in ASA
4. Write Global State
5. Write Local State
6. Remove Global State
7. Remove Local State
8. Approve Transaction
9. Reject Transaction

3.7 Condition

A condition node is a single comparative expression that is used in a Met node, And Conditional or Or Conditional node.

CHAPTER

FOUR

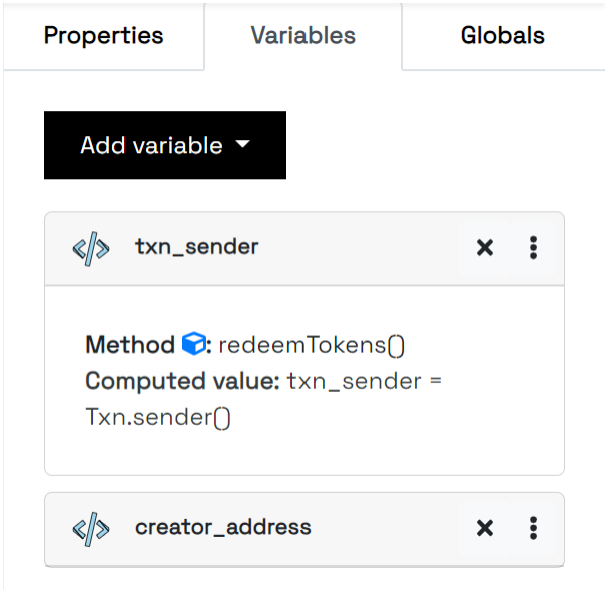
CONTROL FLOW

In every application, we have a way in which logic can be designed. In this sense, ASC Builder can be used to create a logical expression such as an `If` statement which uses comparative operators and performs actions when due according to conditions met or not met.

The structure is thus:

```
Decision Node
├── Met Node
│   ├── Condition Node
│   ├── Routine Node
│   │   ├── Action Node
├── Routine Node
│   ├── Action Node
```

A typical illustration of a decision statement would be to create two variables such as txn_sender and creator_address thus:



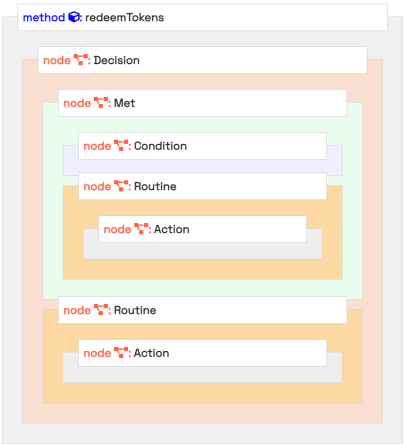
We may want to check if the transaction sender is the creator of the smart contract and approve if that is the case or reject if that is not the case.

To do this, we follow these steps:

1. Create a Decision node

2. Create a Met node and add it to the Decision node.
3. Create a Routine node and add it to the Decision node.
4. Add a Condition node to the Met node, which has the first operand as `txn_sender`, second operand as `creator_address` and the operator as `Is Equal to`.
5. Add a Routine to the Met node.
6. Add an Action to the just added Routine with `Approve Transaction`.
7. Add an Action node to the Routine directly nested in the Decision node with `Reject Transaction`.

The resulting GUI would appear thus:



The structure of a decision **MUST** follow the following pattern.

You can watch the video in this Tweet to see how this is done.

CHAPTER

FIVE

GLOBAL STATES

Static Global States can be specified (to be created upon application creation). A state is created with a Key and the associated value.

There are three forms of Global States that can be created with respect to their data types:

Data Type	Description
Byte String	Used to provide a value of ByteSlice that should be stored in the state.
Integer	Used to provide integral value that should be stored in the state.
Address	Used to provide a valid Algorand address that should be stored in the state.

Upon creation of the Application, the created Global States will be created for use in the rest of the application's lifetime.

CHAPTER

SIX

OVERRIDING DEFAULT METHODS

At default, there are two methods that are defined when a user opts in and when the application is created.

These two methods are `OptIn()` and `OnCreate()` respectively.

The default action in these routes is `Approve()`, when there are no states or definitions to override them.

When a Global State is added (via Global state component), it is due to be created upon application creation.

Once that is set, the application creates them upon deployment.

Nevertheless, the behaviours for the scenarios of opt in and creation can be altered by creating methods, `OptIn()` and `OnCreate()` and adding nodes that should perform the intended functions.

Likewise, the behaviour of the clear Program can be altered by creating a `clearProgram()` method and adding nodes to it.

Once these methods are created, the default `Approve()` return value would be overriden in the Pyteal source code result returned.

CHAPTER

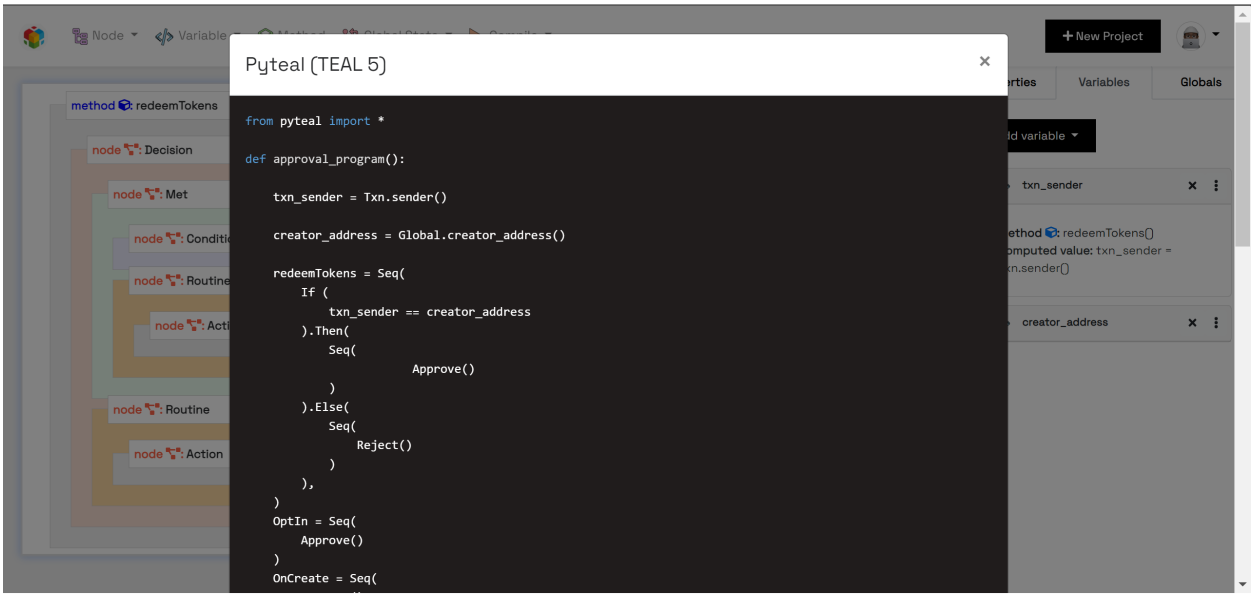
SEVEN

COMPILING

The GUI workflow can be converted to a JSON schema or Pyteal source code.

When the “Export JSON schema” button is clicked, a JSON file is downloaded, called asc-builder.json, which can be imported to create another project with the same data.

A typical Pyteal source code result would appear thus:



The code can be copied to clipboard and then executed to generate TEAL code.