Smart contract development using NeoVM and SmartX – The Basics

1. **Prerequisites**

Before we start with the actual development process, we need to ensure that we have the required tools at hand.

The following are the essentials that will be at the core of the development process of smart contracts using Ontology’s platform

* SmartX – Ontology’s online smart contract IDE and debugger
* Cyano wallet – Google Chrome extension
* Explorer – Web based public tool used to track blockchain activity and transactions in general

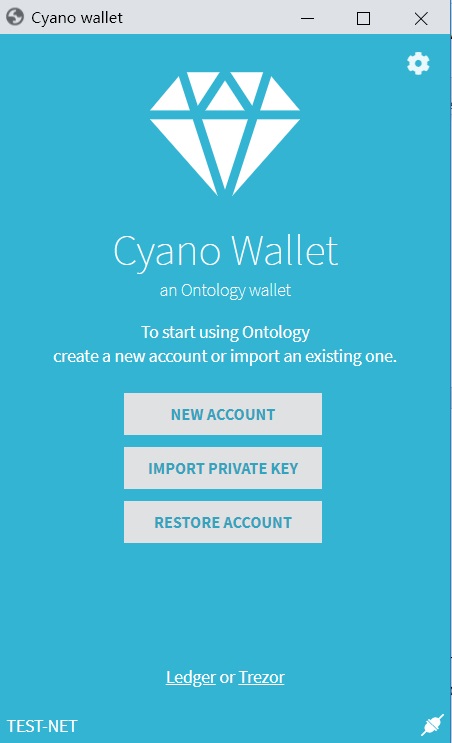
1. **Environment setup**

Setting up the development environment can be a complicated and time-consuming process. That is the reason why we have tried to make this step as swift and convenient as possible. The tools are web based and are practically ready to use right away.

Firstly, for the sake of simplicity and convenience, we will be testing the contracts we develop on the test net, thereby eliminating the need of a private chain. A private blockchain network architecture can be set up just as easily on your local environment using Ontology’s Punica suite, a set of development tools that allows for smart contract deployment and testing on the private net, but that is beyond the scope of this tutorial.

The first tool that we need is a web browser. We recommend using Google Chrome for the entire process as Cyano wallet is a Chrome plugin. Download link - <https://www.google.com/chrome/>

Next, we need to install the Cyano wallet plugin in the browser. You can search Cyano Wallet in the Chrome store, or follow this link - <https://chrome.google.com/webstore/detail/cyano-wallet/dkdedlpgdmmkkfjabffeganieamfklkm>



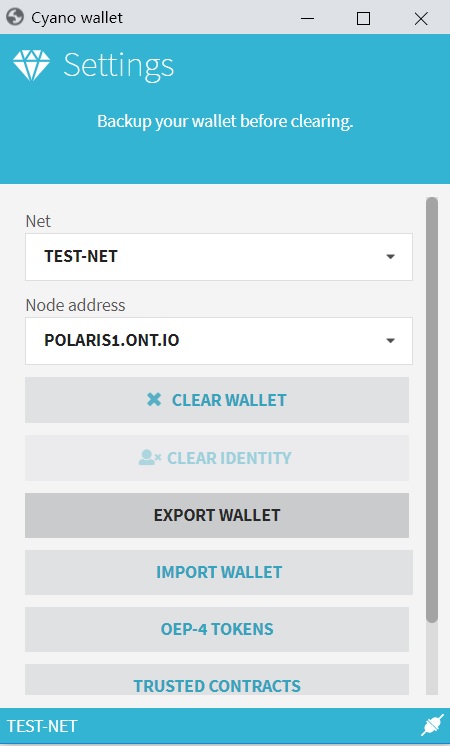
Once the installation completes and you launch the wallet for the first time, you will be prompted login.

In case you already have a Cyano account, you can choose to login using your private key and mnemonics phrase.

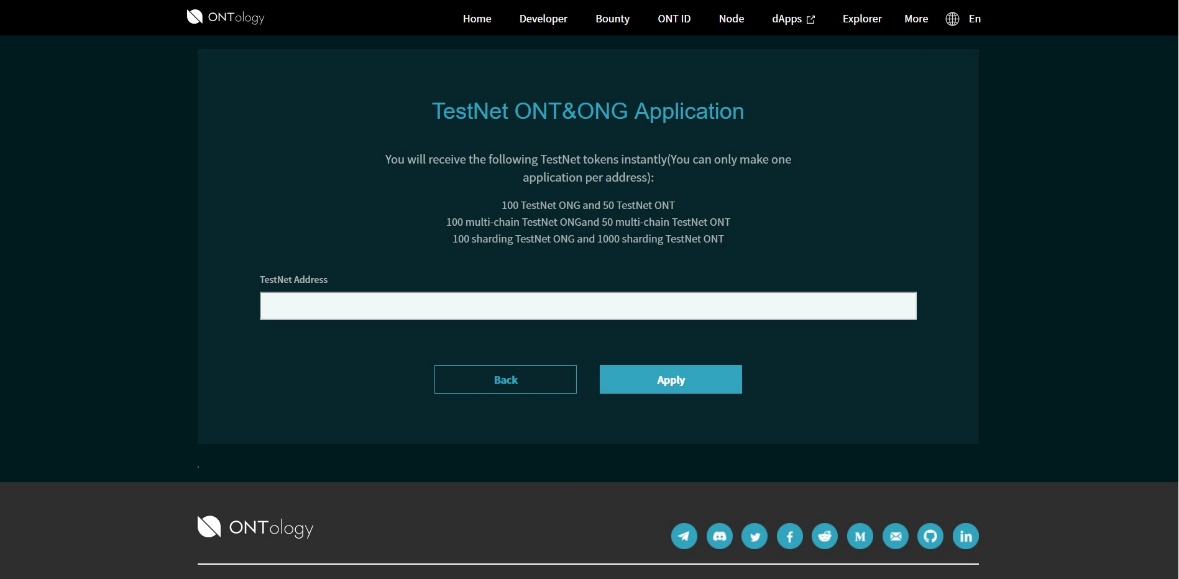
If not, register a fresh account and proceed. Ensure that you save the private key and the mnemonics phrase in a secure location.

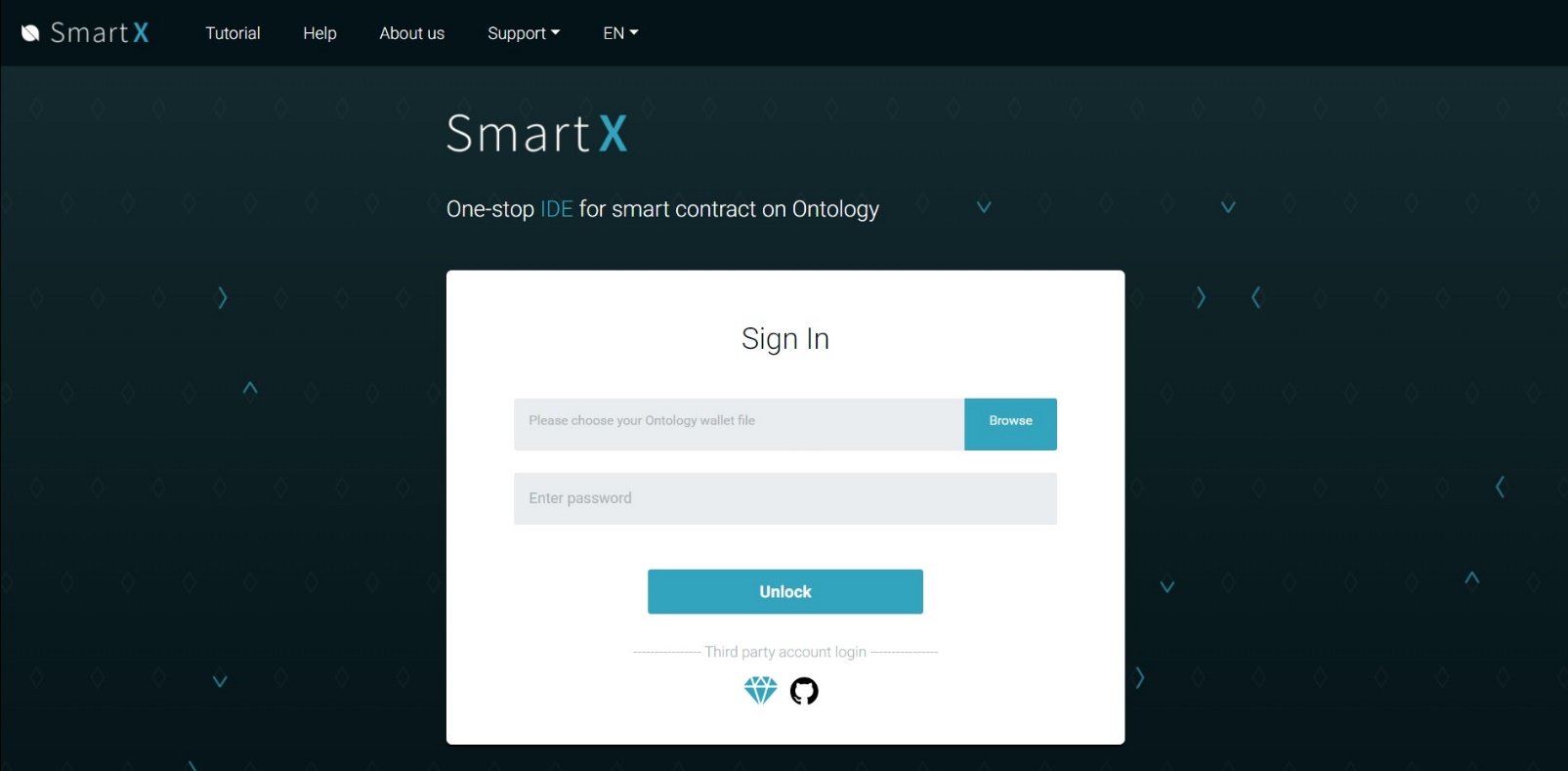
Once logged in, export the wallet to an external **.dat** file. This can be done by accessing settings by clicking on the cog in the top right corner. Ensure that it is stored securely. We will use this file later to work with SmartX.

Next, change the network setting to TEST-NET. We will be deploying and testing our logic on the test net.



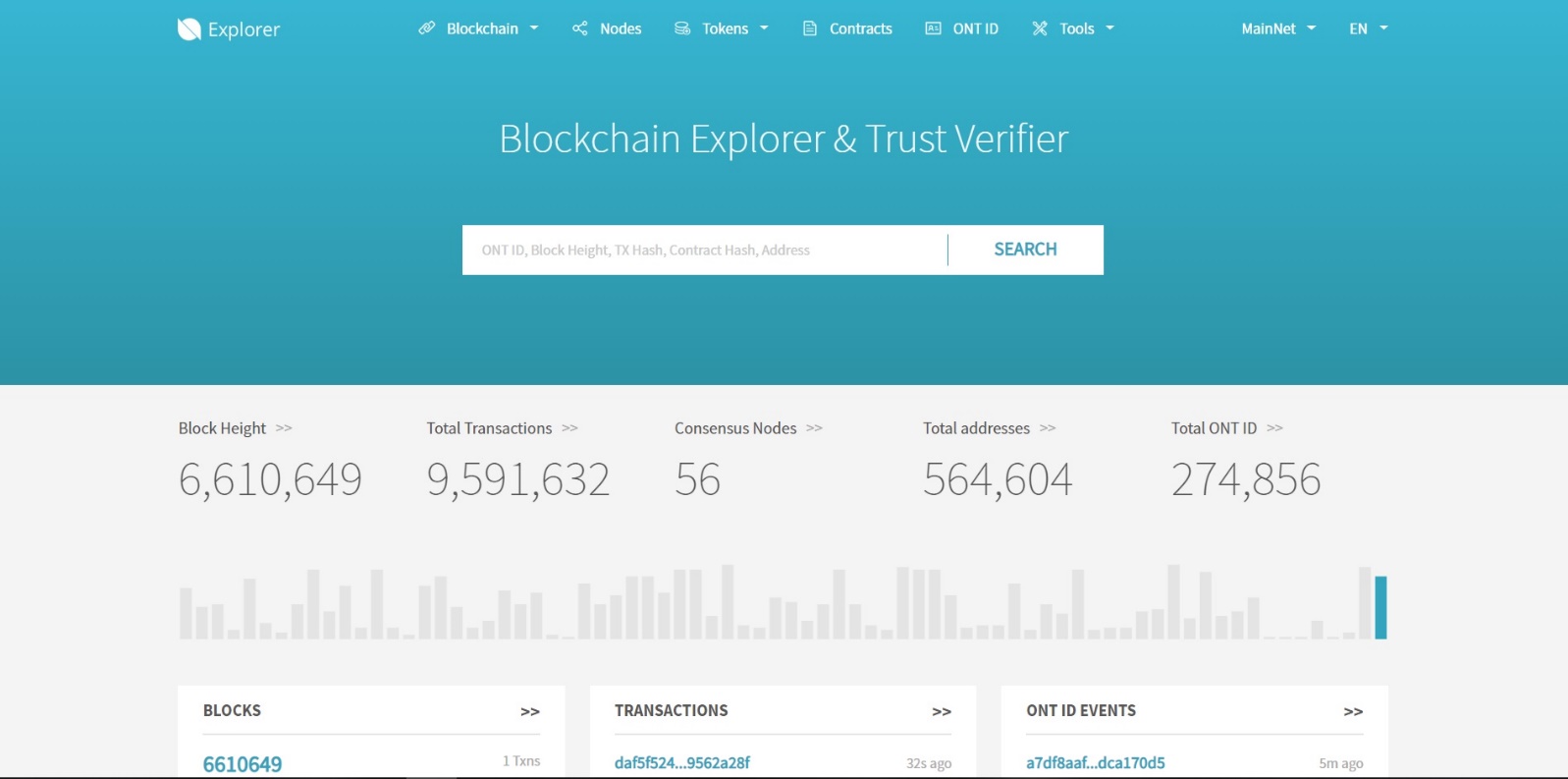
Note: Deploying and testing smart contracts on the test net can be carried out without a mainnet ONT/ONG balance. But, to pay the gas cast of deploying a contract on the test net, you will still need a nominal testnet ONG balance. The gas cost is calculated by taking the product of gas price and gas limit(gas price \* gas limit). Test tokens have been made available for free by Ontology and can be applied for by following given link: <https://developer.ont.io/applyONG>





The IDE that we will be using is Ontology’s SmartX, a browser-based development environment that supports Python, C#, and JavaScript(coming soon). We are going to take an in-depth look at the development process using Python, as far as this tutorial is concerned.

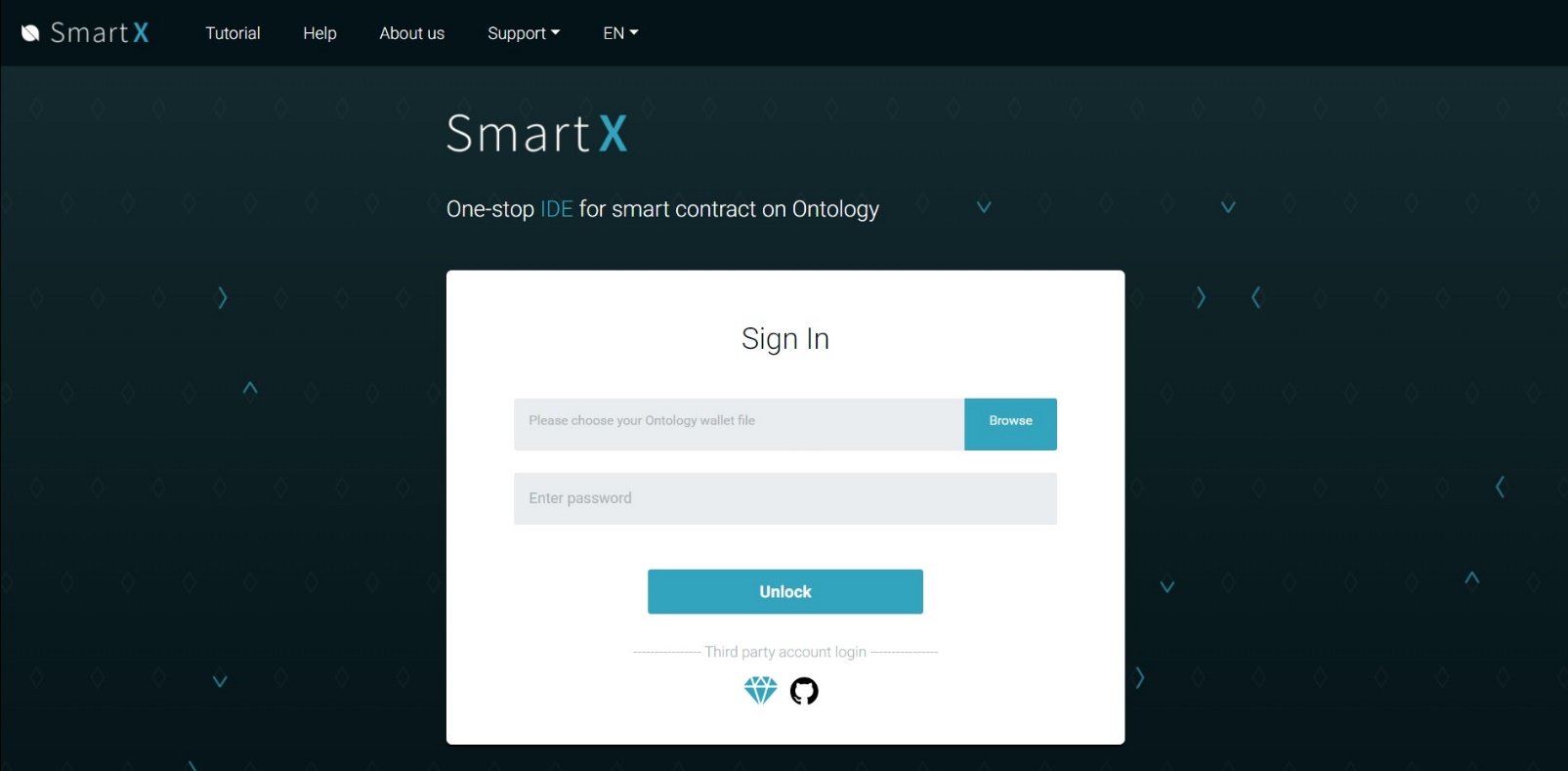
NeoVM serves as the execution engine for the programs written using Python in SmartX. SmartX core integrates all of Ontology’s APIs, and so all the different functions which allow us to perform blockchain related tasks can be used directly by importing the relevant API, which we will be discussing later in this tutorial.

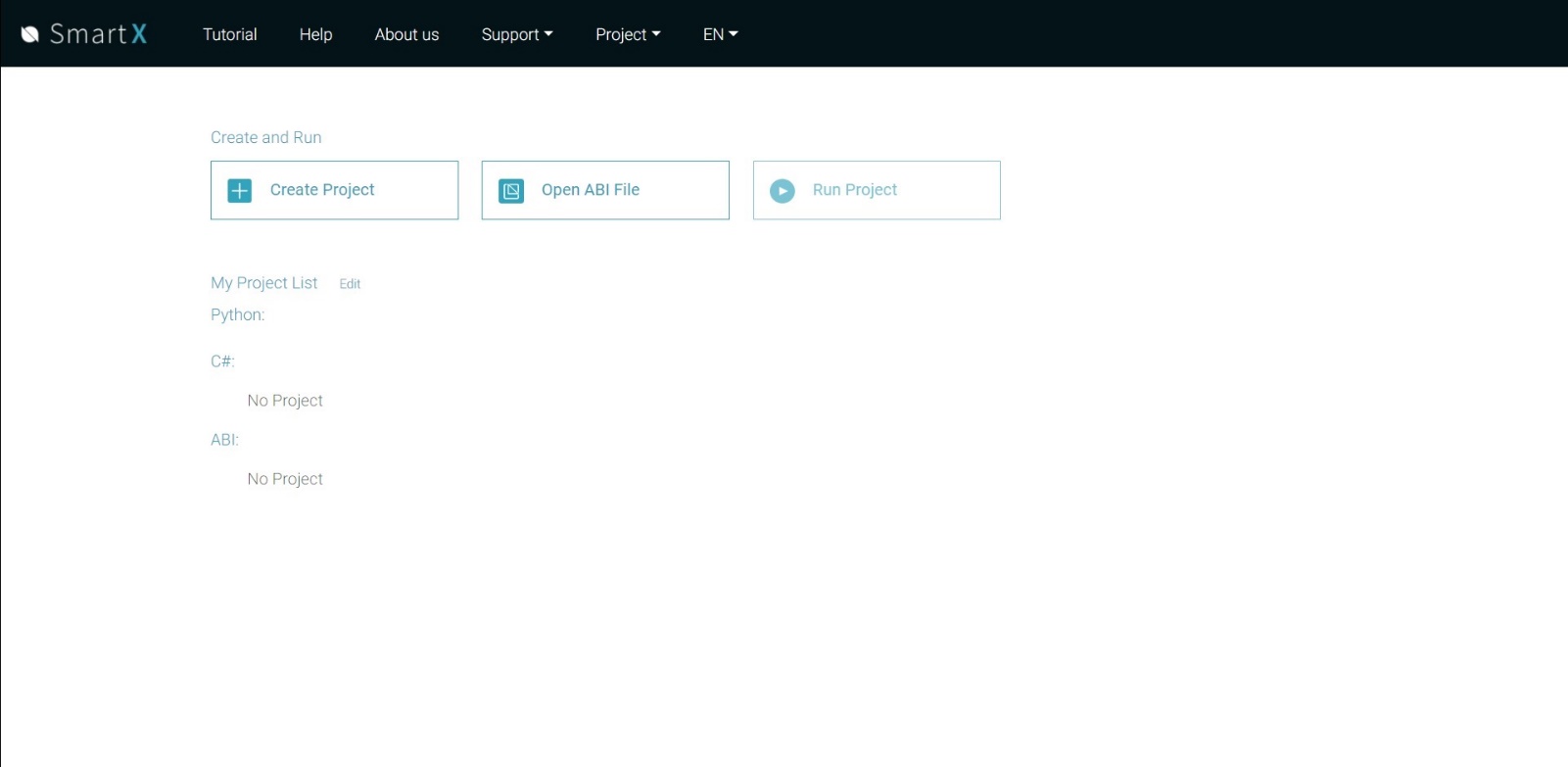


Explorer is an online web based tool that can be used to track transaction and event related blockchain activity. It can be used to monitor both the test net and the main net using transaction hashes, ONT IDs, contract addresses, and even block height.

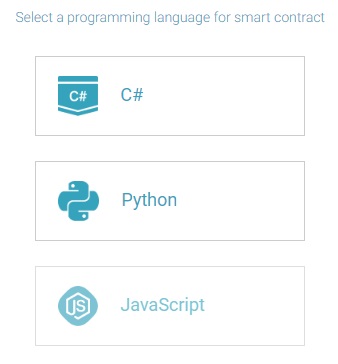
1. **Launching SmartX**

Note: Testing smart contracts requires a Cyano account but does not need ONT/ONG balance.

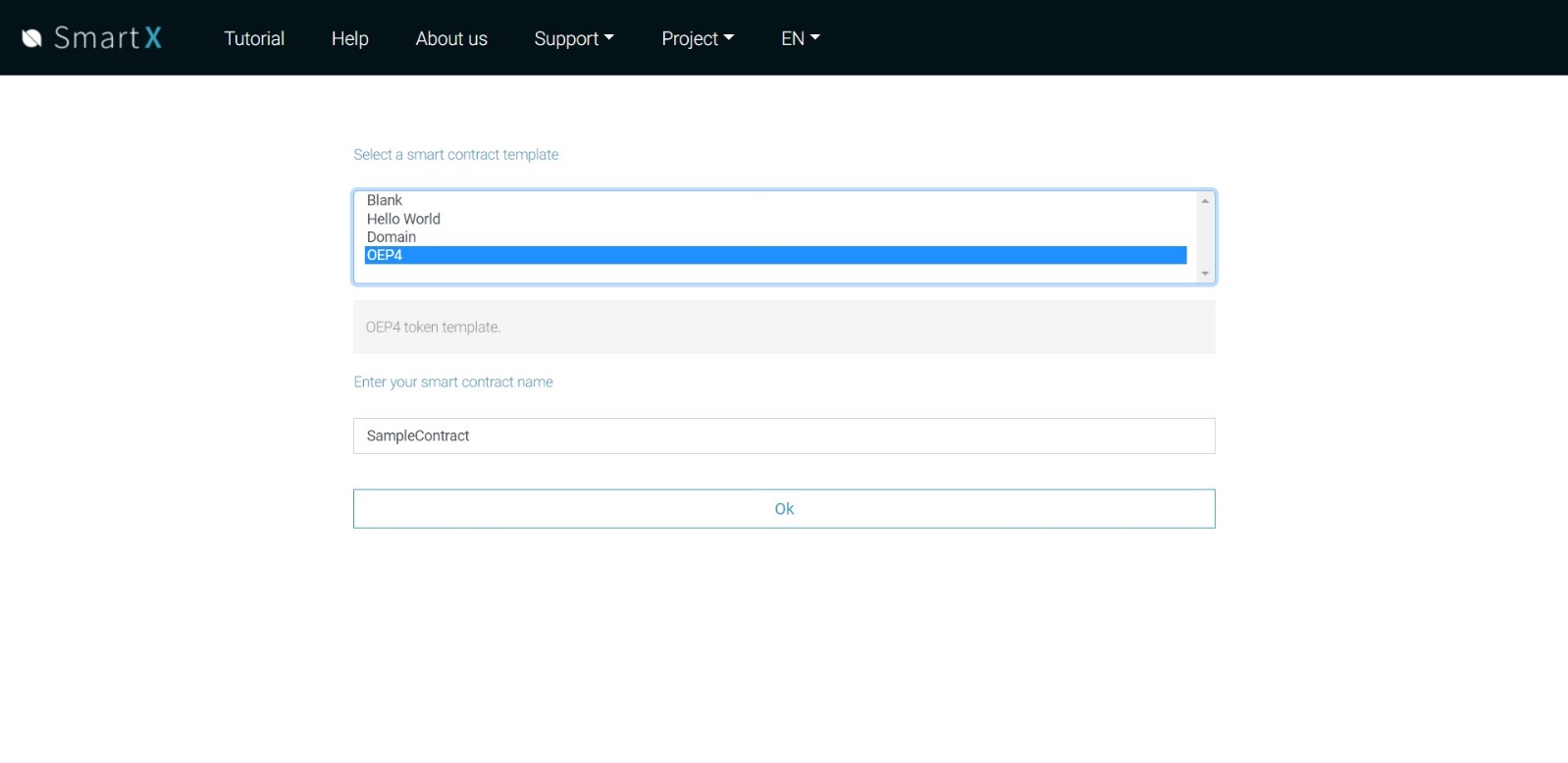




Once logged in you will be prompted to select a project. Create a new one if there are no existing projects.

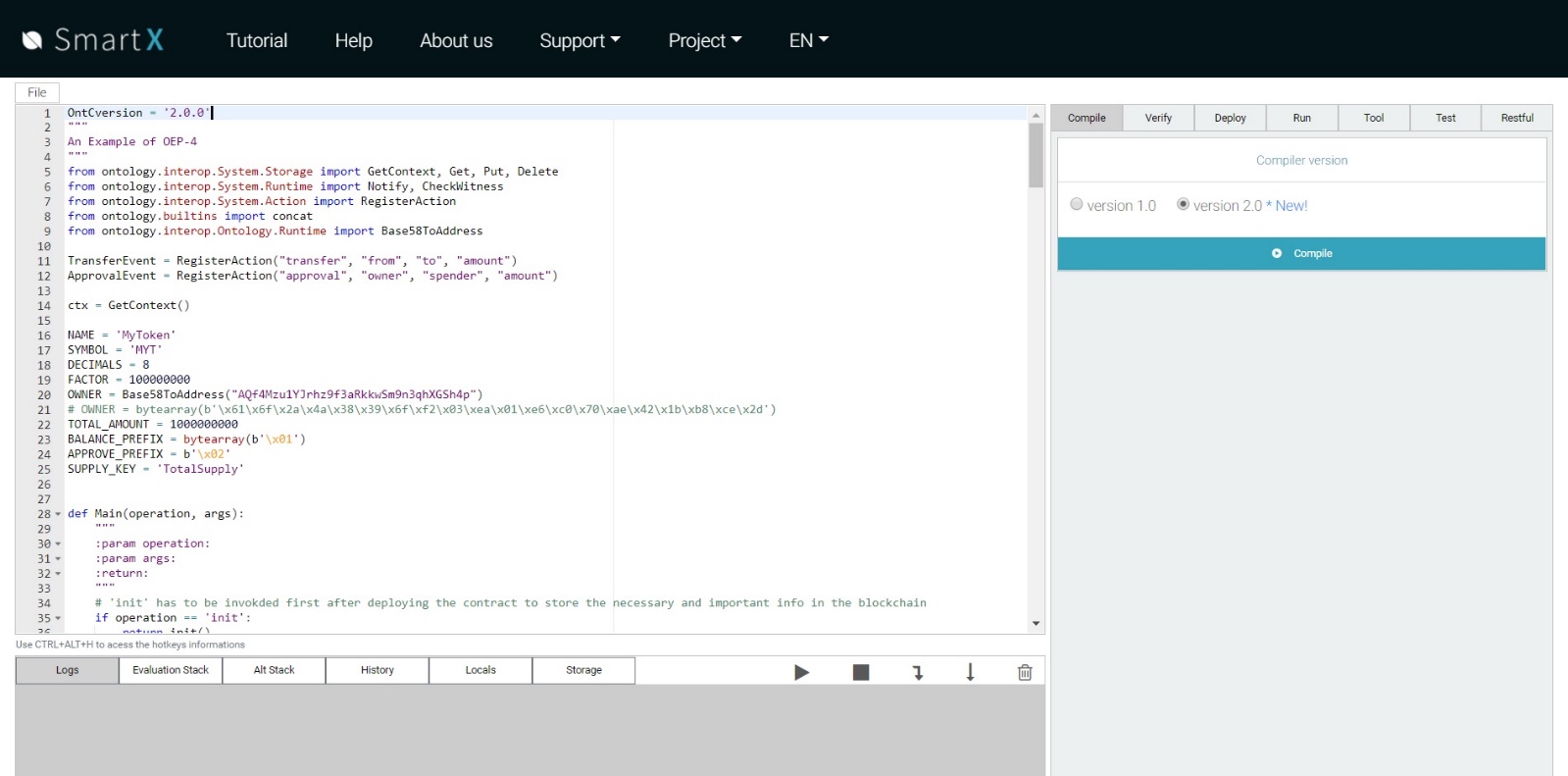


Upon selecting the new project option, you can choose a programming language to work with. In this tutorial, since our focus is on developing smart contracts with NeoVM, we will be illustrating examples and sample code using Python.



Select Python and proceed to the next menu. Here, a list populated with a set of examples and templates of smart contracts will be displayed. You can choose any example and go through the sample code or use it as a base to give structure to your own smart contract logic. For this tutorial we will be looking at the OEP-4 illustration.

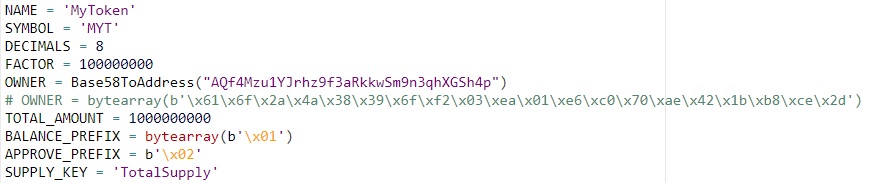
1. **Start writing the program logic**



Once SmartX is up and running, this what the main window looks like.

Since we have selected the OEP-4 template, the code is already present in the editor area. You may choose to edit this code as you please based on the logic that you’re trying to implement. But for the sake of simplicity and staying within the scope of this tutorial we will use the code as it is to ensure uniformity.

It is worth taking some time to take a look at the code and the overall structure of the program that we’ll be working with.



The variables declared in this section of the code define the protocol itself and the specifics that will govern its functionality.

Variables such as “NAME” and “SYMBOL” serve as identifiers for the token.

“FACTOR” is the base 10 value that dictates the precision of amounts that can be transferred. For example, if the value is set to 100, transfer amounts with values up to two levels of precision are supported by the system. “DECIMALS” stores the multiplier value for access convenience.

“OWNER” stores the Base58 address of the entity or account that holds the authority over the totality of tokens and can choose to distribute them as and when needed.

“TOTAL\_AMOUNT” stores the total number of tokens that exist. Always a fixed number.

“BALANCE\_PREFIX” is an access modifier that is used with account addresses for authentication purposes. “APPROVE\_PREFIX” serves the same purpose, but for the approve operation wherein the owner can authenticate another account to use tokens.

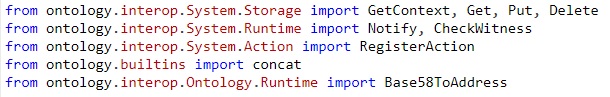
“SUPPLY\_KEY” correlates directly to the total amount of tokens and is used for any operations that may be carried with the total tokens figure, since the value isn’t directly accessible.



This line immediately stands out in the upper section of the code.

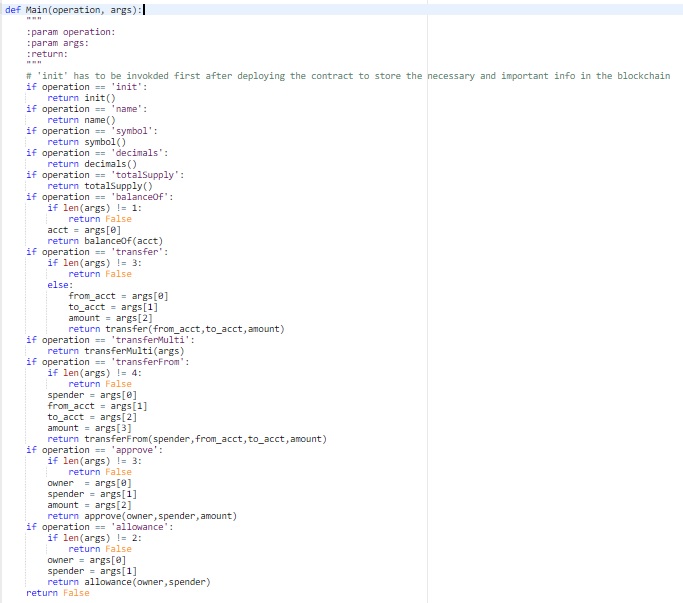
GetContext() is a function that acts as the bridge between the smart contract and the blockchain. It is used when fetching and transmitting data from and to the chain by calling the GET and PUT functions which are a part of the Storage API.

We will go through the relevant APIs as we come across the respective functions, and the full set of available APIs in a later section of the tutorial.



Since we’re working with the online SmartX IDE, all the APIs and functions are available for use and can be accessed directly by importing at the start of the program. Here, the functions that we import are GetContext(), Get(), Put(), and Delete() from the Storage API, Notify(), CheckWitness() and Base58ToAddress() from the Runtime API and the built-in function concat().

Note: In later versions, built-in functions don’t need to be imported and can be called directly.

Let’s take a look at the Main() function.

The Main() function takes two arguments, *operation* and *args*. The *operation* argument is based on the operation to be performed and dictates the function to the executed. The *args* argument helps passing the important information that a function needs to carry out further execution, for example account addresses or input data.

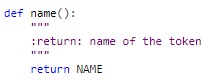
Here, clearly there are 11 different functions that can be called depending upon the argument that is passed in Main(). SmartX passes these arguments using the “Options” pane on the bottom right.

* **init()** **:** The init() function serves as the starting point of the program logic. It initializes the definition variables declared at the top based on the values provided. Thus, this is the first function that needs to be executed post deployment.
* **name() :** This function returns the name assigned to the token, “My Token” in this case.
* **symbol() :** This function returns the symbol assigned to the token, “MYT” in this case.
* **decimals() :** Returns the number of decimal places that designate the precision of valid token values, 8 in this case.
* **totalSupply() :** Returns the total number of tokens assigned while initializing. Denotes the fixed number of tokens allocated for circulation. (Uses SUPPLY\_KEY to fetch the value from the chain, stored earlier during initialization)
* **balanceOf(acct) :** Fetches the corresponding token balance of the account that identifies with the Base58 address passed as argument to the function.
* **transfer(from\_acc, to\_acc, amount) :** Transfers the equivalent token value passed as the amount argument to the to\_acc address from the from\_acc address.
* **transferMulti(args) :** The parameter here is an array that contains the same information, i.e., the sender’s address, receiver’s address, and the amount to be sent, in that sequence at the respective indices. It can be iterated for multiples transactions by passing the respective account addresses and the corresponding amount.
* **transferFrom(spender, from\_acc, to\_acc, amount) :** The spender here takes a certain amount of tokens from the from\_acc address, and transfers them to the to\_acc address.
* **approve(owner, spender, amount) :** The owner authorizes the spender to use a certain amount of tokens from their own account. Both the owner and spender arguments here are Base58 addresses and the amount specifies the amount that the spender is authorized to spend.
* **allowance(owner, spender) :** This function can be used to the amount that the owner account has authorized spender to use. Both arguments in this case are Base58 addresses.

The functions above can be classified into two different types- access functions and utilities. Let us consider the flow of control as these functions are called.

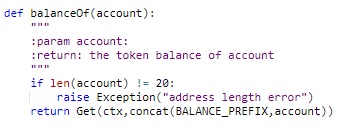
Access functions are primarily used to fetch data post contract deployment. Functions such as name(), symbol(), totalSupply() and balanceOf(acc) allow us to achieve this by using get() function from the Storage API which fetches relevant data from the chain. Let us look at how it is implemented in program logic.

**name() function definition**



This is how a simple access function can be defined. The name() function takes no arguments. Even though functions such as name(), symbol() and decimals() do not explicitly call the get() function, after the contract is deployed, all the data is fetched from the chain.

**balanceOf(acc) function definition**

****

The balanceOf() function takes one argument, a Base58 address which denotes an account. There is a validity check in place that verifies the length of the address and raises an exception if the address is invalid. If the address is valid the get() function is called with two arguments, the account address prefixed with BALANCE\_PREFIX, and the context.

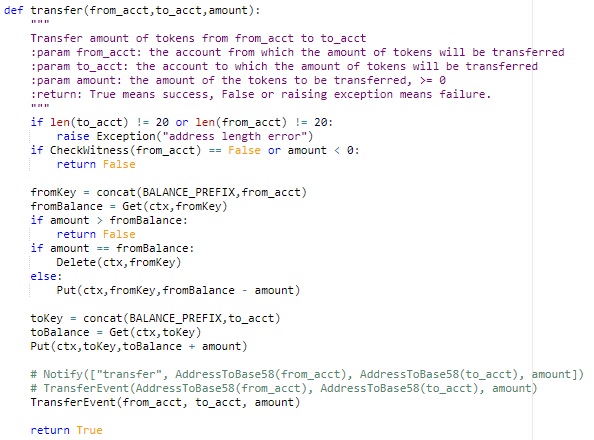
The context allows for data reference on the chain to fetch the account balance value, while the prefixed account address ensures authenticated access. Next, get() returns this data to Main() where it is output to the log window using the notify() function. The totalSupply() function works in a similar fashion.

Note：The balance and approve prefixes are hexadecimal values in ASCII format and can be modified to support your own program logic.

Let us look at the utilities in the sample code.

Utilities are methods that have richer functionality and help modifying the on-chain data, which is the basis for transactions and tasks that may be carried out. The OEP-4 token logic that the code template is using illustrates various use cases and scenarios in the form of functionality. This is to exhibit just how versatile smart contracts are in nature and the different kinds of business logic that they can be used to generate.

**transfer(from\_acc, to\_acc, amount)**



The transfer() function implements the most fundamental transaction feature, transferring tokens from one account to another. It takes three arguments, the sender’s address, the receiver’s address, and the amount to be transferred.

The function carries out verification by a simple length check, but a more complex logic can be developed based on individual needs.

Next, the BALANCE\_PREFIX is concatenated to the sender’s account address, and balance is retrieved by making a get() call using this address. A quick comparison is made in the next step where the sender account’s balance is compared with the amount to be transferred. All three scenarios have been defined clearly.

If the balance is less than the transfer amount, the transaction fails, and the control returns to Main() directly.

If the amount equates to the balance amount exactly, the balance of sender account is set to 0 by calling the delete() method using the sender’s prefixed address. This is practically equivalent to using the put() method to manually assign the value 0 to sender’s account but using put() method in this case might give rise to security vulnerabilities.

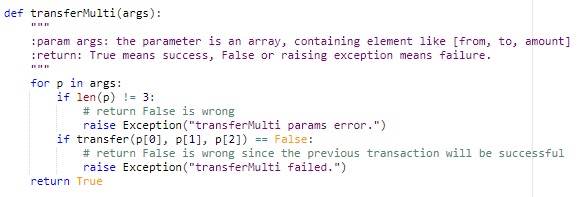
If the balance is higher than the transfer amount, the amount is deducted from the balance by making a put() call and updating the sender accounts balance with the deducted value.

Next, the receiver’s address is prefixed with the BALANCE\_PREFIX, and the prefixed address is used to add the transfer amount to the receiver’s account.

Finally, this transaction event is sent to the chain using the RegisterAction() method for recording in a ledger.

TransferEvent is the alias that RegisterAction() uses here. RegisterAction() is a method of the Action API and it takes four arguments that are transferred to the chain in order to record transaction details. The transaction hash and certain other details are output in the logs section.

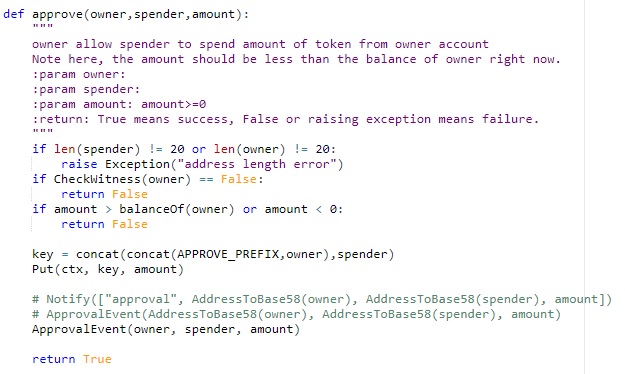
The **transferMulti()** is works in a similar manner. The logic remains the same, since basically all that transferMulti() does is call transfer(), but it allows for multiple transfers to take place simultaneously. It takes one argument which is a nested array.



The sub-array elements are processed in sets of three such that the first and second elements still represent the sender’s and receiver’s addresses, and the third element represents the transfer amount. The sub arrays are iterated till there are no more elements left in the args[] array.

Exception is thrown in case the sub-array does not contain exactly three elements, or the previous transaction fails for some reason, and the control comes out of the loop and goes back to Main()

**approve(owner, spender, amount)**



The approve function implements another complex logic wherein an account, namely the “spender” is given the permission to utilize a certain amount in tokens from another account, namely the “owner”.

The function first carries out address validation in terms of length, and user authentication for the “owner” who is about to perform the approval.

Next, the amount selected for approval is compared to the available balance in the “owner” account.

If the account does not have enough balance the process is terminated, and the control returns to Main().

If the account has enough balance, a key is generated by concatenating the “owner” address prefixed with APPROVAL\_PREFIX and the “spender” address. It is then added to the ledger using the put() method by passing the context, the above generated key, and the approval amount.

The transaction event is then recorded using the ApprovalEvent() method, and the result containing the transaction hash returned by the NeoVM engine is displayed in the logs section.

**transferFrom(spender, from\_acc, to\_acc, amount)**



The transferFrom() function implements a more complex logic and carries out a task that may prove to be useful for certain applications.

This function allows a third party, namely the spender, to utilize a certain amount in tokens that are provided from an account that does not designate to their own credentials, basically implementing the same logic as that of the approve() function. It takes four arguments, which are three Byte58 addresses and one transfer amount.

First, the function carries out the conventional address validation. Then it verifies whether the spender has the authorization to carry out this transaction using the CheckWitness() function which is a part of the Runtime API.

Next, the balance of the “from” account is fetched and cross-checked with the transaction amount to ensure the account has enough balance. The process to fetch the balance remains the same.

The spender’s address is then prefixed with the APPROVE\_PREFIX and the approved amount from is fetched using the prefixed address.

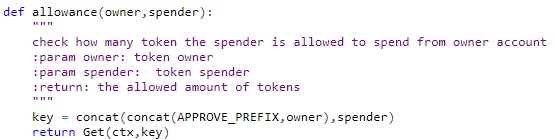
The transaction comes next. If the transaction amount is higher than the approved amount the transaction is aborted, and control returns to Main().

If the amount is exactly equal to the approved amount, the transaction amount is deducted from the “from” accounts balance.

If the approved amount exceeds the transaction amount, the difference is calculated and stored in the ledger for future reference, and the transaction amount is deducted from the “from” account.

The transaction amount is then transferred to the “to” account using the put() function. The event is then recorded and the result with the transaction hash is displayed in the logs section of the IDE.

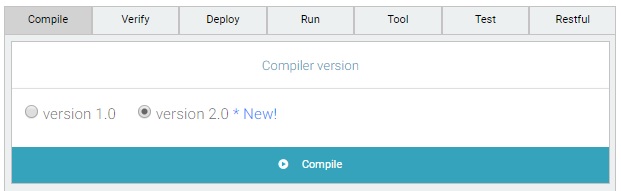
Another function that implements a similar logic has be defined as **allowance(owner, spender)** which facilitates querying the amount of allowance that has been allocated to the “spender” account from the “owner” account.



Practically speaking, this function cam be classified as an access function too in the sense that it returns the allowance value. But it also performs a get() query to fetch this result from the chain.

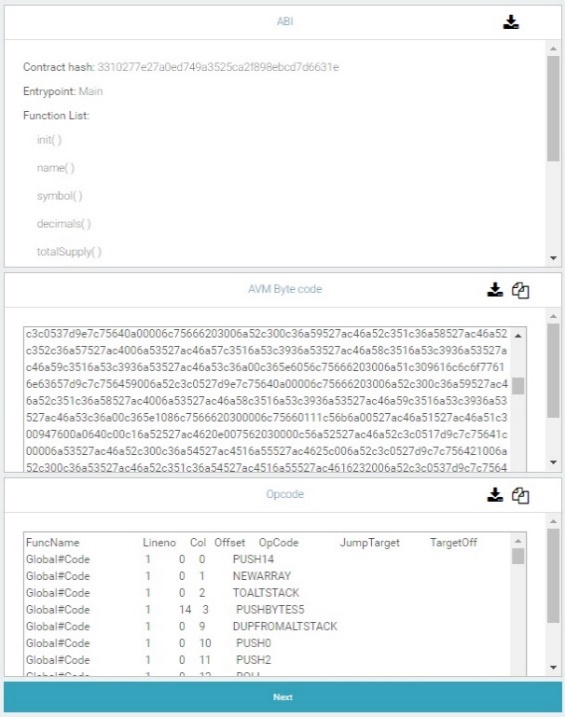
A key generated by concatenating the prefixed owner address and the spender address is passed to the get() method along with the context to fetch the required allowance value, which is then returned to Main(). The value can then be displayed or used to perform other tasks.

1. **Deployment and testing**

****

Once the logic development process completes, we can proceed to compiling our smart contract.

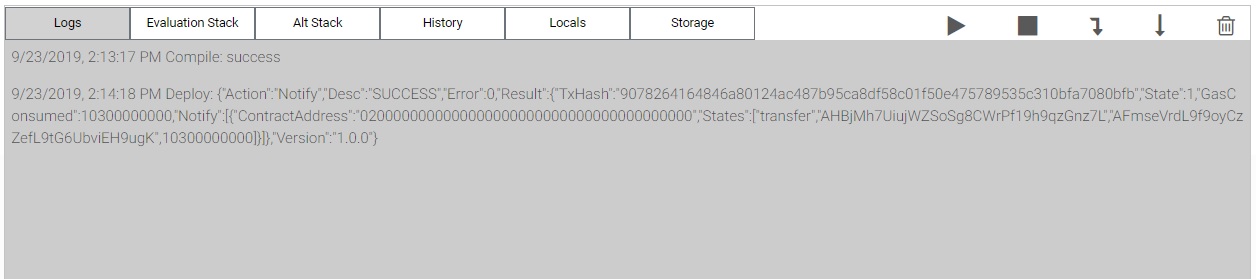
Upon compiling the smart contract, the following results can be seen in the IDE-



In the compile tab, the AVM byte code is the resulting intermediate code produced after compilation. NeoVM processes this AVM code to execute our contract.

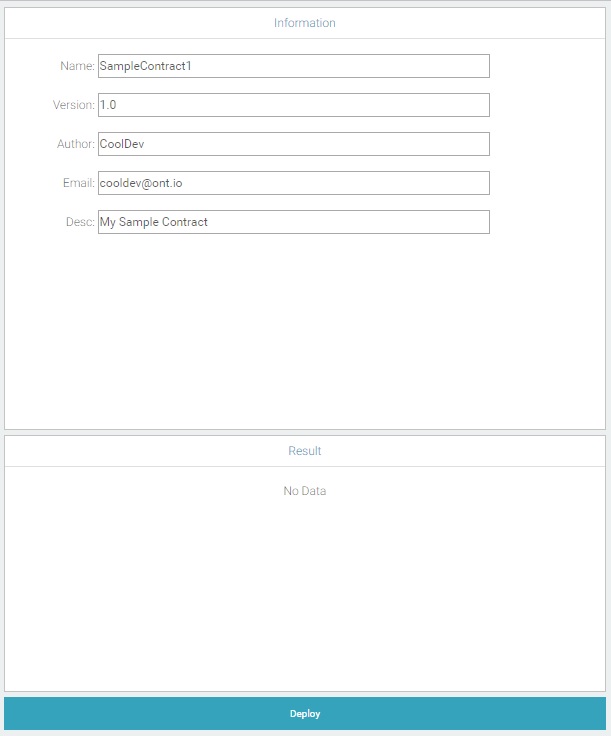
The opcode indicates the stack status line by line; an advanced debugging tool.

ABI stores the parameter information for all the functions and the contract hash itself. The contract ABI can be saved for future retrieval.



The logs section displays the compilers response which includes everything from debugging results to the information that the VM returns.

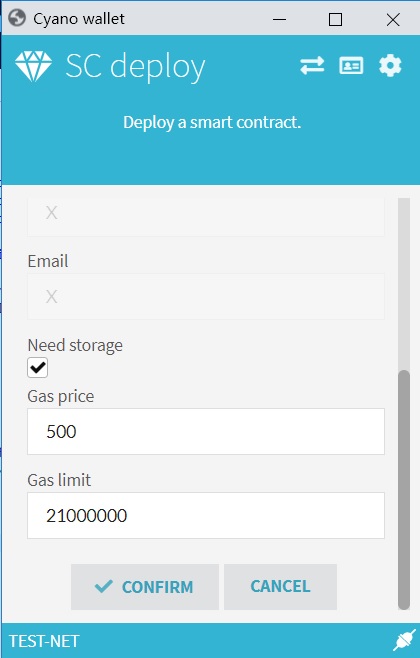
After all the compilation errors have been dealt with and the contract is successfully compiled, we can proceed to deploy it.



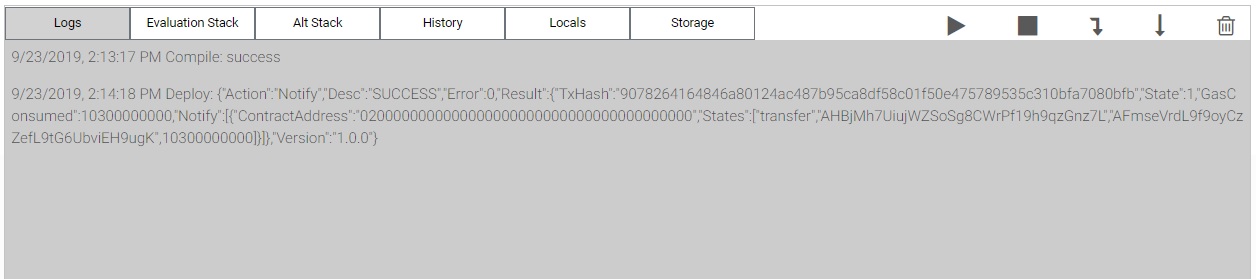
Here, we fill in the relevant information regarding the contract and proceed. A confirmation window pops up where you can enter the gas price and gas limit.

**Note: The gas limit is precise to 9 decimal places. Thus, 109 units of gas would be equivalent to 1 ONG token with the minimum valid value being 0.000000001.**

**The wallet automatically sets a suitable limit based on the complexity of the code being compiled and run. But you always have the option to set a limit yourselves. Ensure that limit is higher than the cost, otherwise the contract may fail to deploy or invoke.**



Once deployed successfully, the transaction hash will be displayed in the logs section and in the result pane in the bottom right.



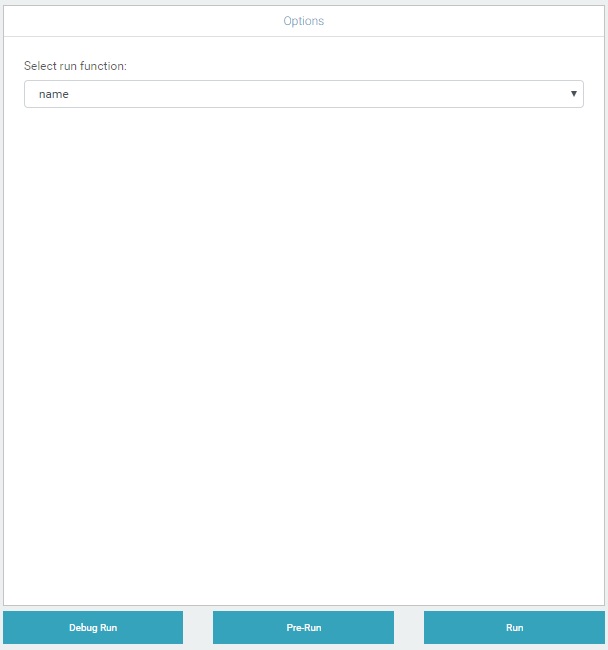
Next, we can proceed to running the contract.

Before executing other functions, we must first initialize the wallet using init() so as to ensure that it has enough balance to carry out transactions.

We can then choose the function that we want to execute from the drop-down menu.

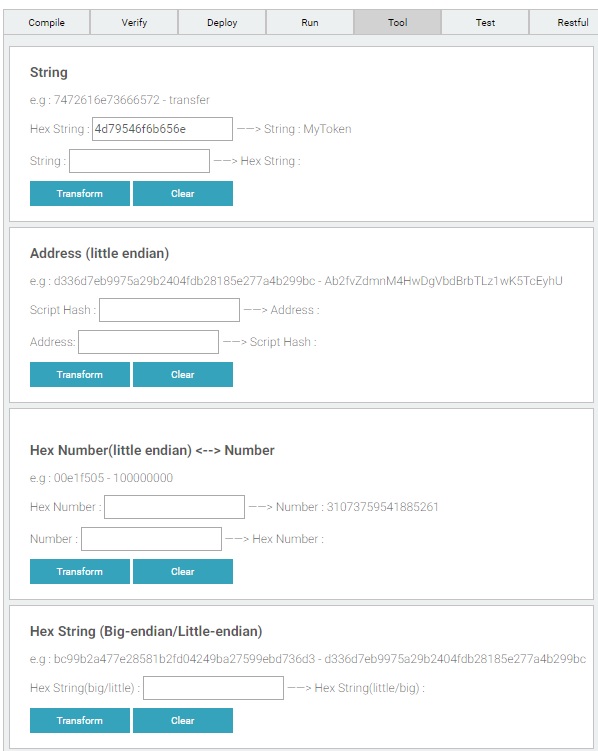
At this point we have two options, pre-run and run. We can choose to directly run the contract, and the engine would run the AVM code generated before. Pre-run is an option which can be chosen to test run the contract to check if it runs as expected. There is no gas cost associated with pre-running a contract.

We can select the function that we want to execute from the drop-down menu, select the data type of the value to be passed and pass the argument in the blank field.

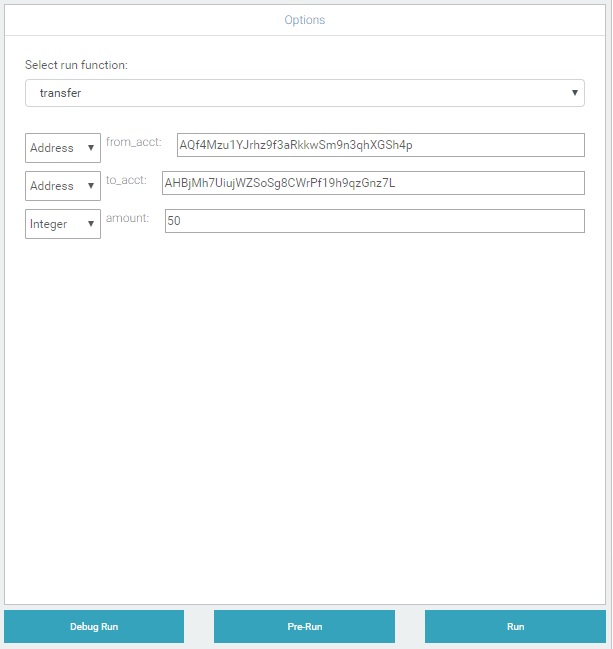




All the results that are displayed in the logs section are in hexadecimal format. The tool option in the right-side pane provides several different conversion tools that are at the user’s disposal which can be used to perform data conversions and a few other functions.



Every time a transaction is executed, a transaction hash will be returned that can be used to track the results. Let us try another transaction, this time with actual tokens being transferred.



Once you run the contract, you will be prompted to confirm the transaction and enter the gas price and gas limit that will used to calculate the gas which the contract consumes in its execution process. (ONG)

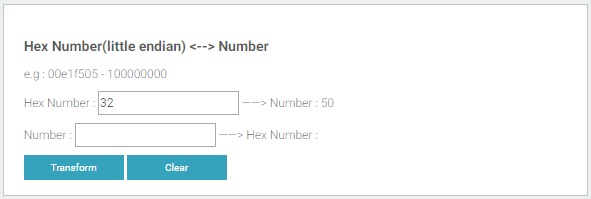
Enter the gas price and gas limit and provide the confirmation. After the transaction is carried out successfully the transaction hash will be displayed in the logs section.



To see the current balance of the account that we transferred the tokens to, we can pre-run the balanceOf() function. We pre-run it because we want to see the output of the function. Running it directly would execute it, but we will not be able to see the value returned.

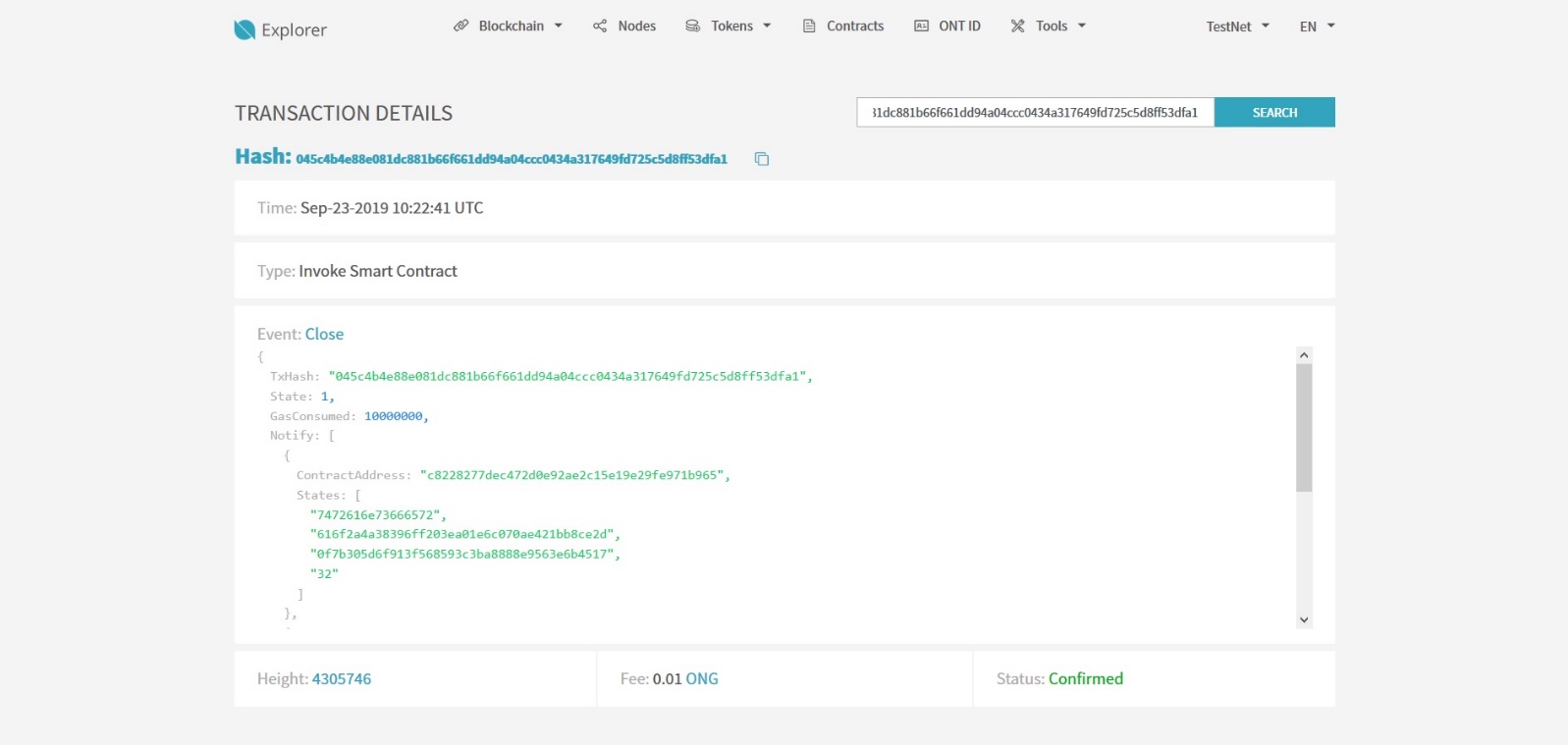


Clearly, 50 units of our sample token have been transferred to the receiver’s address. The displayed value 32 is the hexadecimal value.

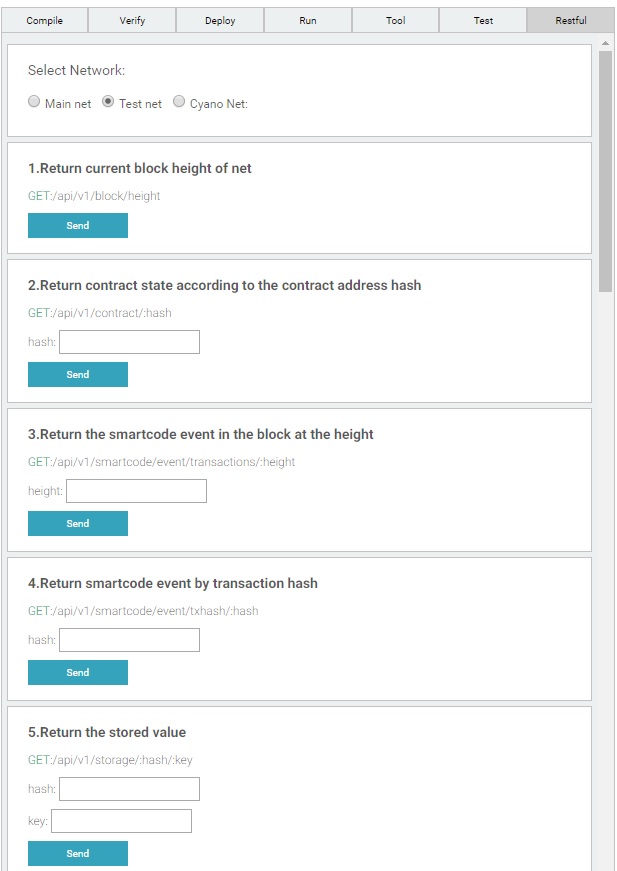


We can convert the hex value to decimal using the converter provided in the tools section.

The transaction hash can be used to see the transaction details in the Explorer.



The last section in the right-hand pane, Restful, is the API that is used to communicate with the chain to fetch useful information regarding the chain’s status, such as the current block height or the smartcode for an event. Smartcode is basically a JSON based representation of smart contract information. The JSON is generated and returned based on the query, i.e., the function called in the Restful API.



The transaction hash can be used to fetch the smartcode for the respective transaction.

The Restful API can be accessed from SDKs too, but SDK integration is another tutorial all by itself.

