# Solidity Smart Contracts 2 and deploying them onto a blockchain

February 9, 2020

**Abstract**

In this activity, we will advance our understanding on how to how to write smart contracts and deploy them onto a real blockchain. To do so, we will use a standard smart contract that creates tokens.

## 1 Smart Contract Standards

To organise Solidity smart contract design, a few standards have been created. Perhaps the most popular is the ERC-20 standard for smart contracts that create fungible tokens, which can be found here:

https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

It lists the required functions and events for smart contracts implementing this standard. An ERC-20 smart contract, is required to: (a) allow for the creation of tokens; (b) keep a list of address balances; (c) allow an address to *transfer* tokens to another address; (d) allow for an address to *approve* another address to take tokens from it at some point in the future; (e) keep a list of *allowances* which detail how many tokens each address has been approved to take from another address.

Of course note that even if smart contracts implements this standard, there can be a variety of implementations of the functions, including implementations that are purposely malicious.

## 2 Exercise: Creating an ERC-20 Smart Contract, deploying and interacting with it

For task 1, we will create an ERC-20 smart contract. For task 2, we will deploy this smart contract to an Ethereum testnet using the MetaMask plugin we learned about in week 2.

*Task 1 - Creating the ERC-20 Smart Contract:* Your task for this activity is to create an ERC-20 smart contract in Remix[1]. To do so, you will take the provided smart contract template code and complete it by filling in the sections marked with square brackets, i.e.: [...]. To complete the contract correctly, follow these instructions:

1. *Variable visiability* - decide what visiability type (e.g. public, private, internal) should be placed on the following global variables in order to abide by the ERC-20 standard: name, decimals, symbol and totalSupply.

2. *Constructor* - complete the constructor according to the notes in it's comments.

3. *balanceOf function* - decide what global variable to return for this state query function.

4. *transfer function* - Note that this function begins with a require statement that requires the balance of the sender to be greater or equal to the value he is trying to transfer. If this require statement is not satisfied, the function invocation will be reverted. After this require statement, you need to correctly code the updates to the contract variables so that this token transfer is stored correctly. Finally, the ERC-20 definition states that the transfer event must be emitted after a successful transfer, so provide the correct variables into the event emit.

5. *approve function* - decide how to record this approval using the provided state variables and emit the Approval event as required by ERC-20. Note if an address X approves another address Y to take tokens, this action should not change X or Y's balance until address Y decides to take this approved tokens (via the transferFrom function below).

6. allowance function - decide what global variable to return for this state query function.

7. transferFrom - This function should only be invoked after the approve function. Therefore the require statement is used to enforce this. Decide how the line previous to the require statement should be coded to enforce that an address cannot transfer more value than he has been approved to do. Finally decide which event should be emitted from this function to abide by ERC-20.

Compile and test your contract to make sure the functions work as expected. I.e: (a) balances of each address can be checked; (b) an address can only transfer tokens if he has that many tokens; (c) an address can approve another address to take tokens (without affecting their balances); and (d) an address can take tokens he has been approved to take (but no more than that number).

[1] https://remix.ethereum.org

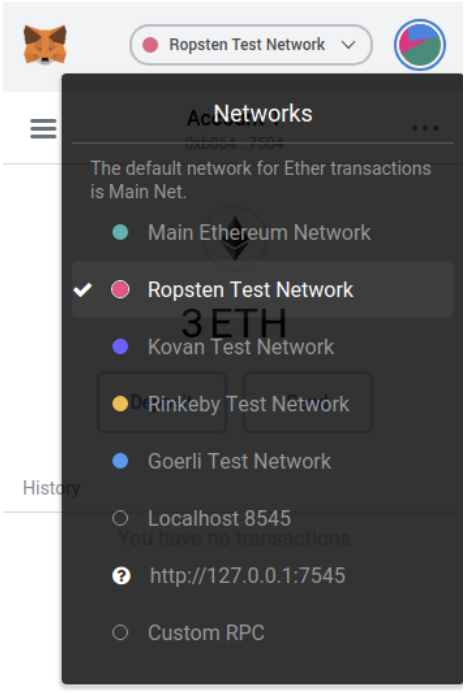


Figure 1: Connecting to the Ropsten test network via MetaMask.

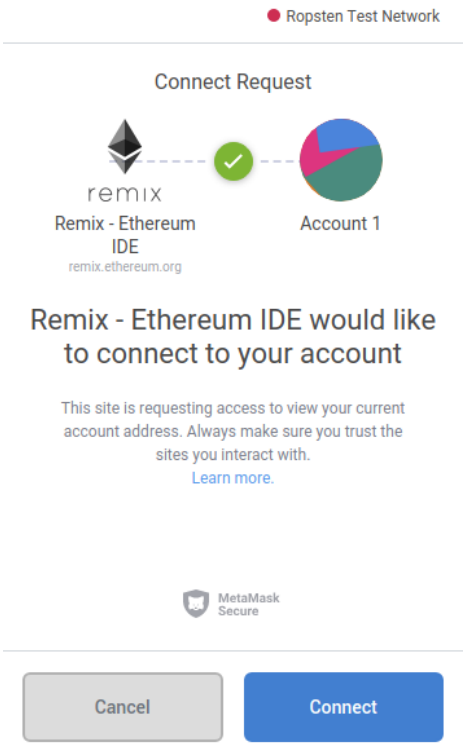


Figure 2: Remix requesting connection to MetaMask.

*Task 2 - Deploying your Smart Contract:* Before we deploy this smart contract onto an Ethereum testnet[2], it is assumed that you have completed the MetaMask activity from week one. That is, it is expected that you are using the Chrome browser, have the MetaMask Chrome plugin installed, MetaMask is connected to the Ropsten testnet and you have gathered some test Ether from the Ropsten faucet. If you have not completed this activity, please return to it before continuing.

For this task, we will be using the Ropsten Etheruem testnet. For clarity, we will firstly check that the MetaMask plugin is connected to the correct network. Click on the MetaMask plugin (i.e. the fox head icon) and login. Make sure that the network displayed is Ropsten, otherwise click on the network tab and select it, like shown in Figure 1. Make sure that you still have some Ropsten test Ether available, otherwise go and get some more from the Ropsten faucet, as discussed in week one.

Now to deploy the smart contract onto the Ethereum Ropsten testnet via Remix, move to the 'Deploy and Run Transactions' plugin. Change the environment tab from Javascript to 'Injected Web3'. This should trigger a Metamask

[2]Note that there are a few Ethereum testnets

pop up, where you should see a request from Remix as shown in Figure 2 - allow this connection. Notice that allowing this connection means that Remix can load your real account into the accounts section of the 'Deploy and Run Transactions' plugin. Now simply click on the 'Deploy' button to send a contract creation transaction to the Ethereum Ropsten testnet. This will trigger MetaMask asking you to confirm the transaction, as shown in Figure 3. Click on the confirm button and you will see the transaction pending - MetaMask is waiting for this transaction to be added to the blockchain. Once the transaction is confirmed, MetaMask will allow you to see details on the transaction, such as displayed in Figure 4. In Figure 4, notice the button with the 'View on Etherscan' label. If you click on that you will be taken to the relevant page of Etherscan (an Ethereum blockchain explorer) that displays information on your transaction. The Etherscan transaction details page includes a 'To:' section, which displays your deployed contract address - *you will need this contract address next week*. If you click on your deployed contract address you will see further information on your contract. We recommend that you verify and publish your source code so that you get into good practice on providing transparency of your distributed applications. To do so, move to the contract tab of your contract's webpage on Etherscan, as shown in Figure 5 and click on the 'Verify and Publish' link. Your verification options will include the 'Solidity (single file)' compiler type and the compiler version must match the compiler listed in the 'Solidity Compiler' plugin of Remix. Once you have successfully verified your smart contract on Etherscan, the Solidity code will now be displayed in the contract tab of your deployed contract's Etherscan webpage.

Finally for this task, we will briefly mention that you can use Remix to interact with an already deployed smart contract as long as you have the Solidity code. To do so, get the smart contract address, go to the 'Deploy and Run Transactions' plugin, select the correct compiled code, paste the contract address into the 'At Address' textbox and click on the 'At Address' button. You will see that a new Remix instance of the contract will appear that can be interacted with - investigate the differences between state query and state change functions now that the smart contract is on testnet.

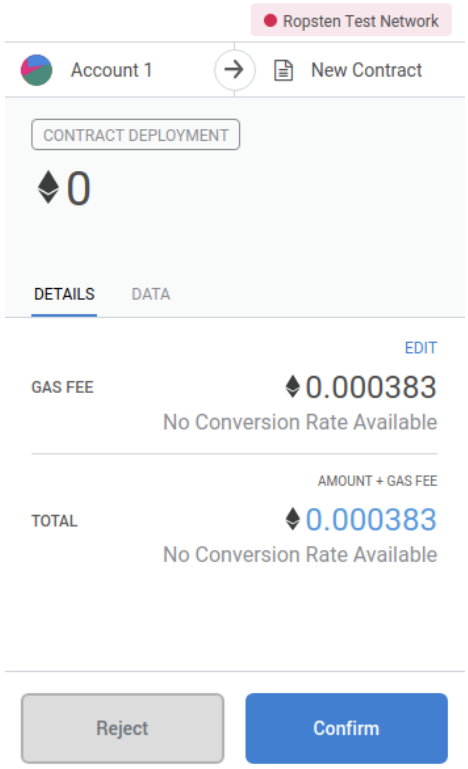


Figure 3: Deploying a smart contract via MetaMask.

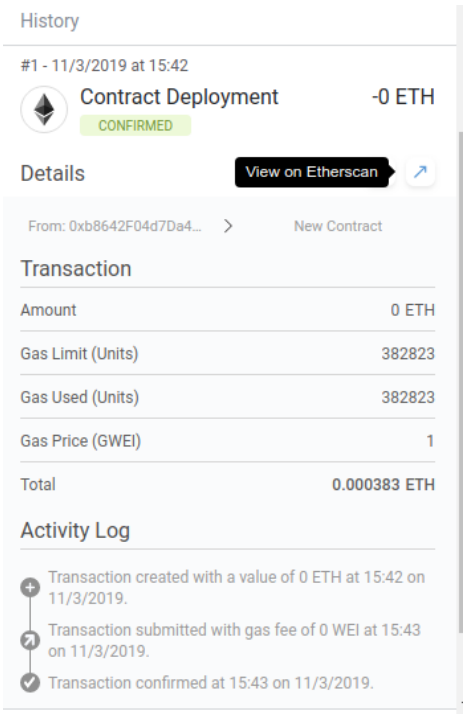


Figure 4: MetaMask's transaction history display.

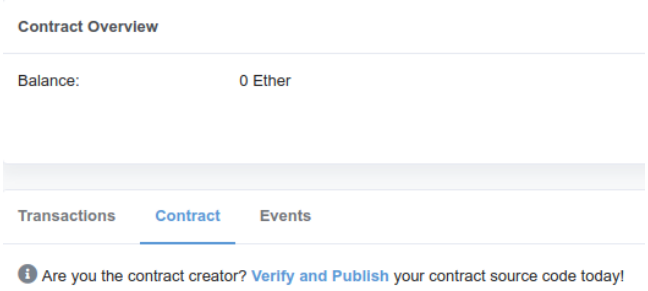


Figure 5: Etherscan's verify and publish contract section.