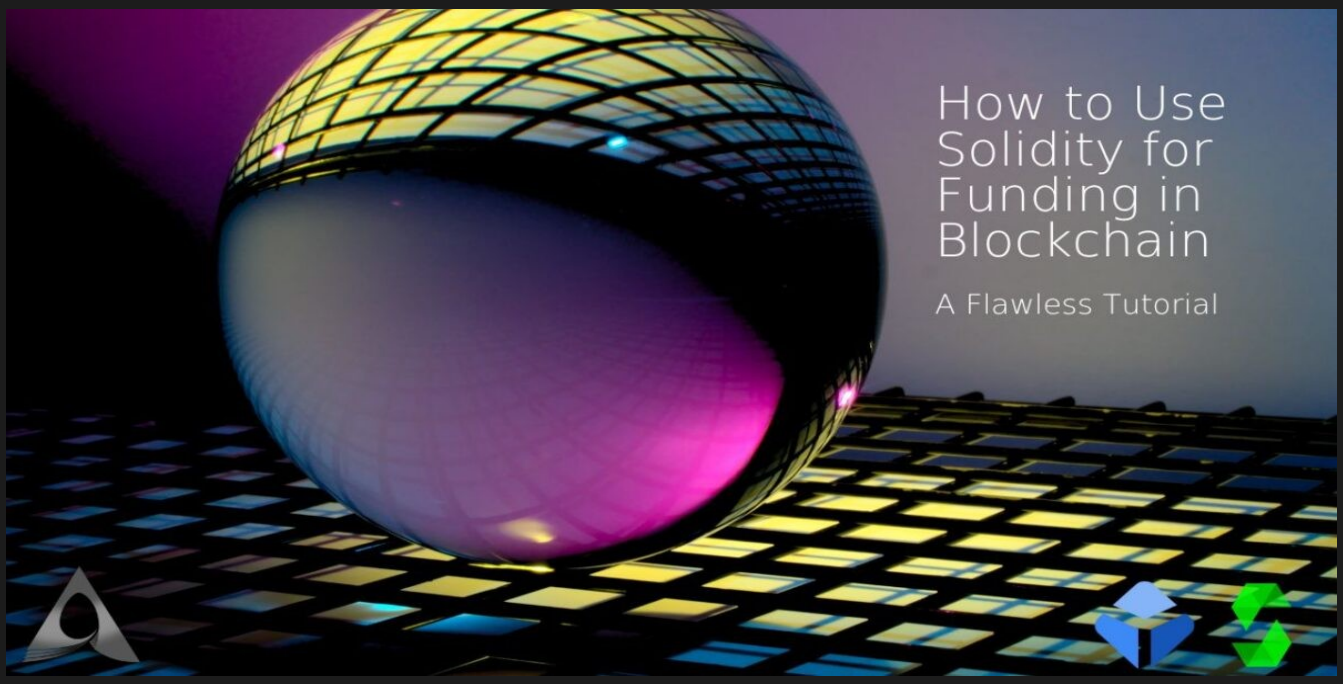
Title	HOW TO USE SOLIDITY FOR FUNDING IN BLOCKCHAIN: A FLAWLESS TUTORIAL
Description	Tutorial
Date	June 02, 2022
Author	Arashtad
Author URI	https://Arashtad.com



This article aims to introduce Solidity language as a tool to write smart contracts. For the ease of the audience, it begins with Remix IDE to write the Solidity scripts. In this article, we are going to see how data types are defined in Solidity and learn how to work with Remix IDE. In addition to that, we are going to write a very simple smart contract to store and retrieve data. Besides, we're going to show how to use Solidity for funding in Blockchain.

USING SOLIDITY FOR FUNDING IN BLOCKCHAIN

If you are familiar with the term ICO (acronym of Initial Coin Offering), you know that it is a fabulous opportunity for startups. They can crowdfund using blockchain tools such as Ethereum smart contracts and offer a coin or a token for the first time to the investors. So, learning to crowdfund on Ethereum blockchain using Solidity will be vital if you are about to start your fantastic journey venturing into the world of smart contracts.

GETTING STARTED WITH THE FUNDING CONTRACTS:

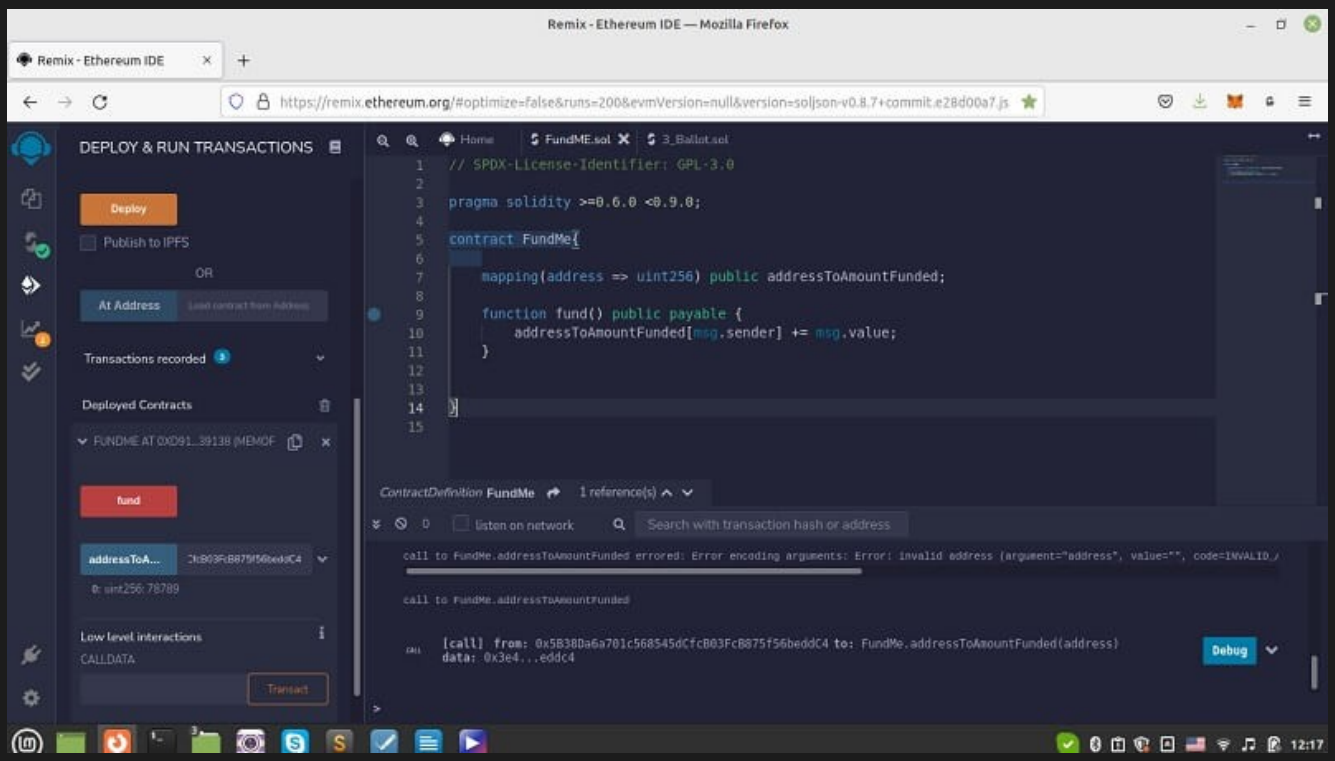
We start our first project named FundMe.sol. By using this script, we are going to keep track of who sent us some money.

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >= 0.6.0 < 0.9.0;

contract FundMe{

        mapping(address => uint256) public
addressToAmountFunded;

    function fund() public payable {
        addressToAmountFunded[msg.sender] += msg.value;
    }

}
```

In the above code, we have defined a mapping from the address of the sender to the value that is sent. Then, we define a public function that is payable with the “fund” name. A payable modifier for the fund() function will ensure that certain conditions are met before executing any transaction. In our function, we have mapped the amount of the sent Ether to the address of the sender.

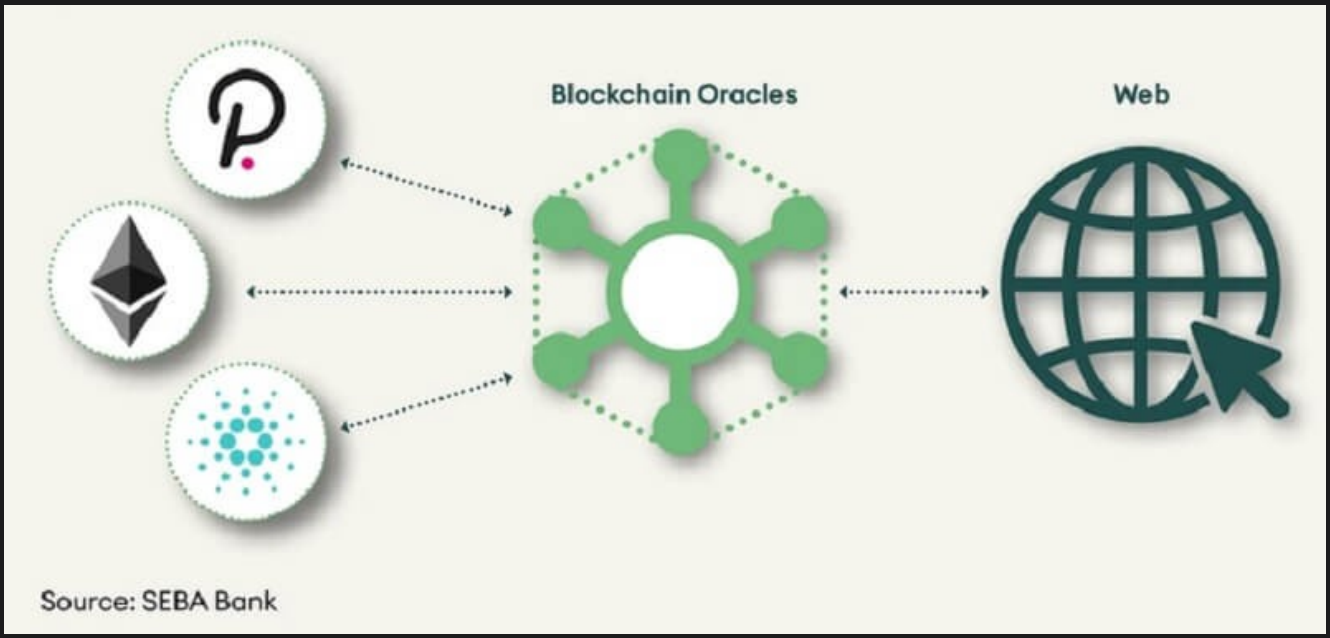
USING SOLIDITY FOR DEPLOYING THE FUNDING CONTRACT IN BLOCKCHAIN

Once we deploy the above contract, enter a value in the value section and press the fund key. Then, we will be able to see that a transaction has been executed successfully. And, if we paste the address of the successful sender in the addressToAmountFunded section, we will be able to the amount that we entered in the value section. And that means our code successfully shows the amount of the sent Ether.



CALCULATING THE DOLLAR EQUIVALENT: THE ORACLE

Now, one question; “we know the amount of Ether, but how do we know how much dollar has been sent for us?” You might suggest using coin market cap API for the conversion of the value, but there is a more sophisticated way to do this. And that is fetching the price directly from the blockchain. Actually, that is why we need oracles. In Blockchain, an oracle is a third-party service that feeds information from the outside world to the smart contracts and vice versa. Here to use Solidity for funding in Blockchain, we need a sort of a back and forth interaction between prices and the transactions for the cryptocurrencies. That is why oracles have come to our help. Notice that Blockchain oracles are decentralized as opposed to other oracles that are centralized.



In the project, we use Chainlink oracle to feed the data of Ethereum price to our smart contract. To do that, copy the scripts below to another file in the current folder and name it AggregatorV3Interface.sol.

```
// SPDX-License-Identifier: MIT
pragma solidity >= 0.6.6 < 0.9.0;

interface AggregatorV3Interface {
    function decimals() external view returns (uint8);

    function description() external view returns (string memory);

    function version() external view returns (uint256);

    // getRoundData and latestRoundData should both raise "No data
present"
    // if they do not have data to report, instead of returning
unset values
    // which could be misinterpreted as actual reported values.
    function getRoundData(uint80 _roundId) external view returns (
                uint80 roundId,
                int256 answer,
                uint256 startedAt,
                uint256 updatedAt,
                uint80 answeredInRound
                );
```

```
function latestRoundData() external view returns (
                uint80 roundId,
              int256 answer,
              uint256 startedAt,
              uint256 updatedAt,
              uint80 answeredInRound
              );
     }
```

After that, import it in the Fundme.sol file by writing:

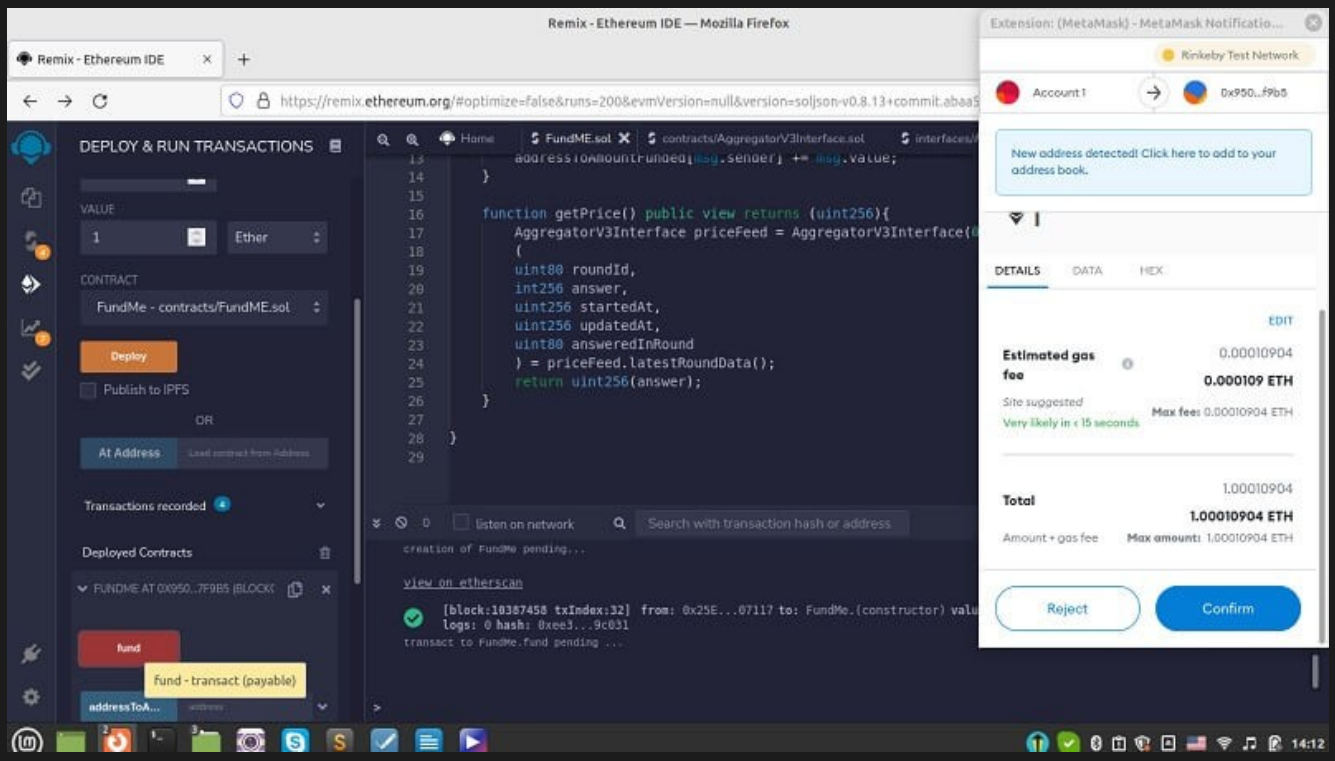
```
import
"@chainlink/contracts/src/v0.6/interfaces/AggregatorV3Interface.sol"
;
```

We also add a function in our contract to be able to call the latestRoundData function on AggregatorV3Interface.sol contract and retrieve the ETH/USD price from the Rinkeby network. So, our code becomes like this:

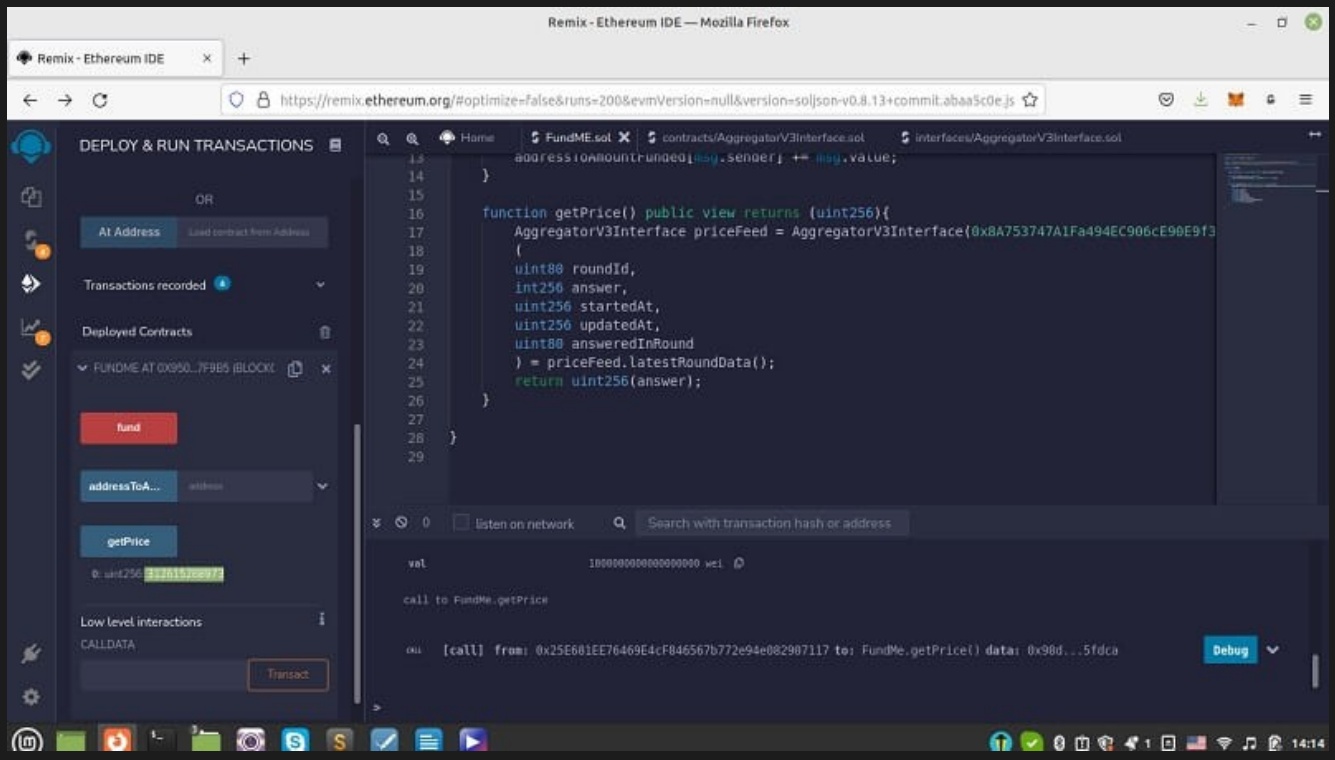
```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >= 0.6.6 < 0.9.0;

import
"@chainlink/contracts/src/v0.6/interfaces/AggregatorV3Interface.sol"
;

contract FundMe{
     mapping(address => uint256) public addressToAmountFunded;
     function fund() public payable {
          addressToAmountFunded[msg.sender] += msg.value;
          }
```

```
    function getPrice() public view returns (uint256){
        AggregatorV3Interface priceFeed =
AggregatorV3Interface(0x8A753747A1Fa494EC906cE90E9f37563A8AF630e);
        (
        uint80 roundId,
        int256 answer,
        uint256 startedAt,
        uint256 updatedAt,
        uint80 answeredInRound
        ) = priceFeed.latestRoundData();
        return uint256(answer);
        }
}
```

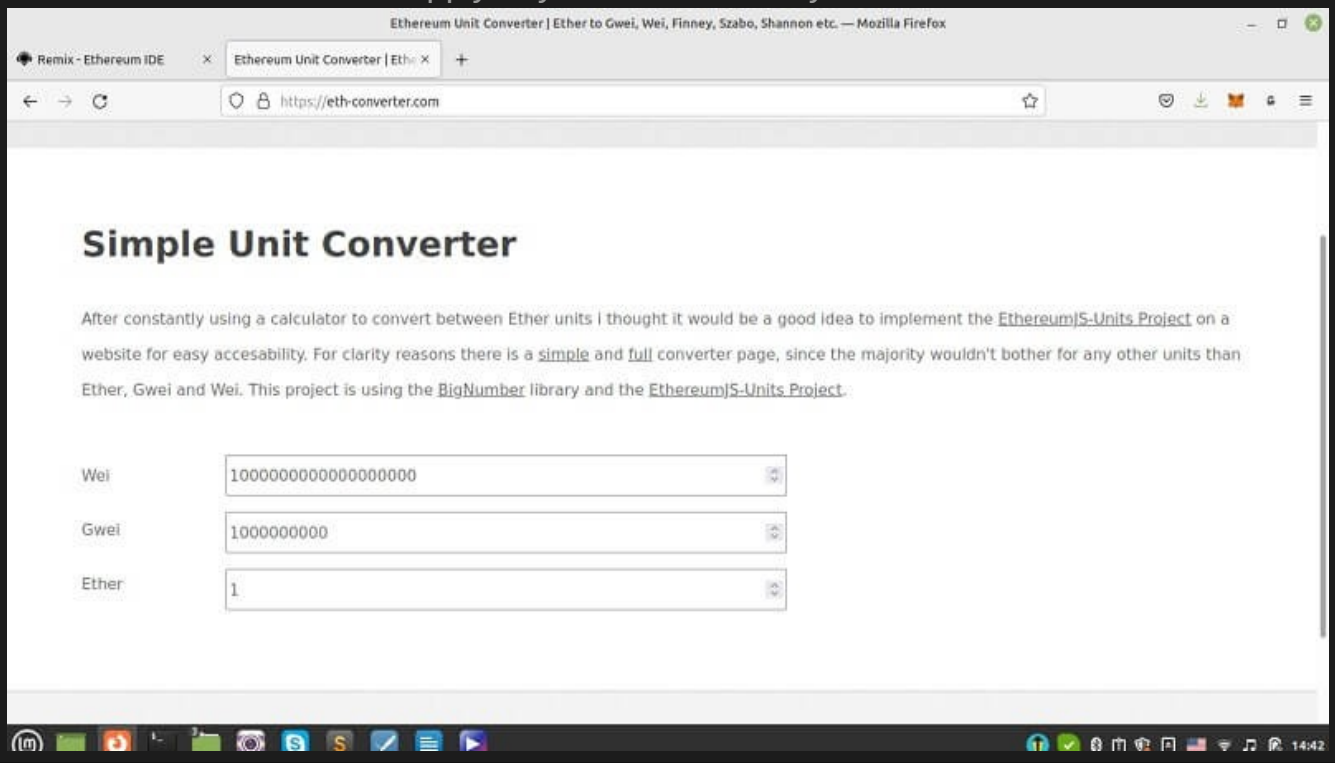
In this updated version of our code, getPrice function tries to read from the imported contract. We have copied the ETH/USD Rinkeby version pair price from Chain Link Docs and pasted it in the AggregatorV3Interface() to be able to retrieve the price from Rinkeby test net. Finally, we have returned the answer with the conversion of uint256 as it was declared as int256 instead uint256. The output of this function must be the amount of Ethereum funded in USD. So, we compile and deploy our contract on injected web3 and confirm the Metamask popup. Then, in the value section, we enter 1 and next to it, instead of Wei we choose Ether. After that, press the fund key and again confirm the Metamask popup window:



Now, after the transaction has been completed successfully, we press the getPrice button. Then, we will see that the amount of Ether we have received appears in USD:



The output is 312615268973, but the actual Ether price is 3126. 15268973. So, why do we have the price multiplied by 10 to the power of 8? The reason is that the price was not in Ether but in Gwei. If you want to convert Ether to Gwei or Wei, you can check out " Ethereum Unit Converter" and apply any conversion that you need.



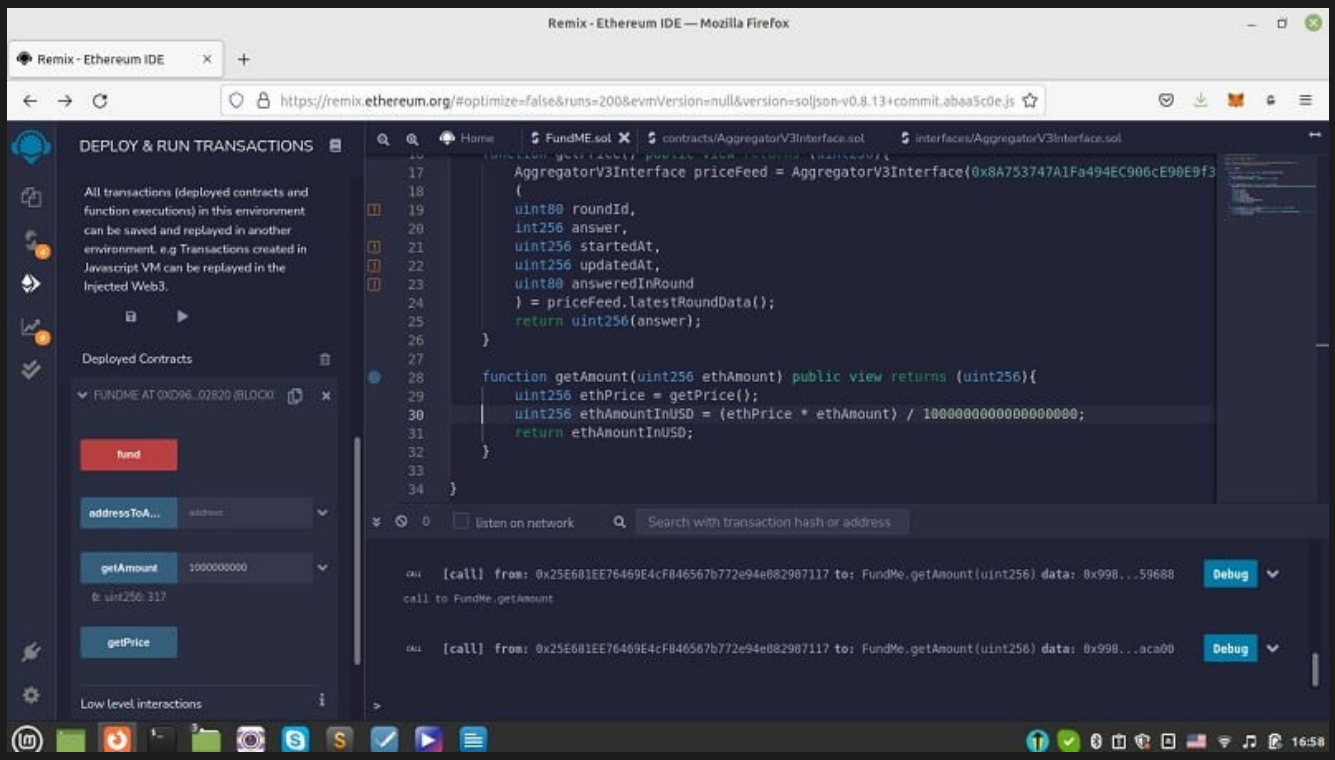
We can also multiply the output by the conversion rate get the actual price in USD as the output of the FundMe contract.

A COMMON PROBLEM IN USING SOLIDITY FOR FUNDING IN BLOCKCHAIN

Now, we are going to cover a common problem in most versions of Solidity and that is data overflow. To solve this problem, we are going to use chainLink safemath. We are also going to see how we can withdraw all the funds as the admin of the fundme.sol smart contract. All of these steps are going to be taken inside the Remix IDE.
In the previous section, we learned how to find the dollar amount of Ether. Besides, we understood that the output of the prices could be in Gwei or Wei meaning that they need to be divided by a number. The following function will do that for us for Wei input values:

```
function getAmount(uint256 ethAmount) public view returns (uint256)
{
     uint256 ethPrice = getPrice();
     uint256 ethAmountInUSD = (ethPrice * ethAmount) /
1000000000000000000;
     return ethAmountInUSD;
}
```

The photo below shows the result of this conversion:



OVERFLOWS IN SOLIDITY

Notice that Since we are dealing with some great numbers especially when we use Wei or Gwei, there is a pitfall that any programmer might face and that is overflow. Suppose you have defined a variable to be uint32 and your number ends up being equal or greater than 2 to the power of 32 which is 65536. And here is when overflow occurs.
When an overflow happens our number, in this example, starts over from zero, for example, if uint32 x equals 65537, it will equal 1. However, In the newer versions of the Solidity (from Solidity 0.8 on), this event will be checked. There are also some libraries that cover this problem and check the overflow problem if happened in the lower versions of Solidity like 0.6.0. One of these libraries is SafeMath and if you are using lower versions of Solidity, you can import it using the below script:

```
import "@chainlink/contracts/src/v0.6/vendor/SafeMathChainlink.sol";
```

So if we want to use this library for uint256 inside of our contract, we write:

```
using SafeMathChainlink for uint256;
```

Notice that libraries in Solidity are similar to contracts. They are deployed only once at a specific address and their code is reused. Inside the context of a contract, if we want to apply a library A to a certain type B, we use the following phrase:

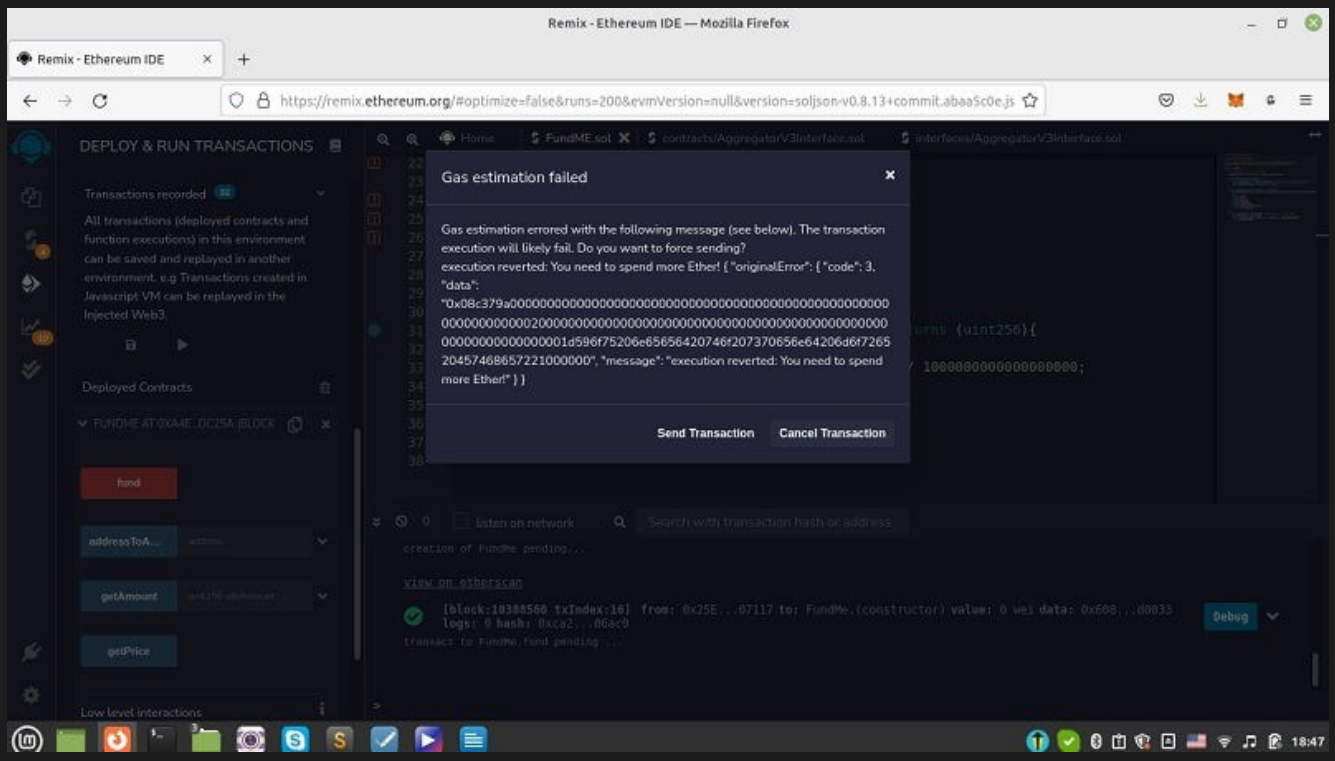
```
using A for B;
```

Require() Function

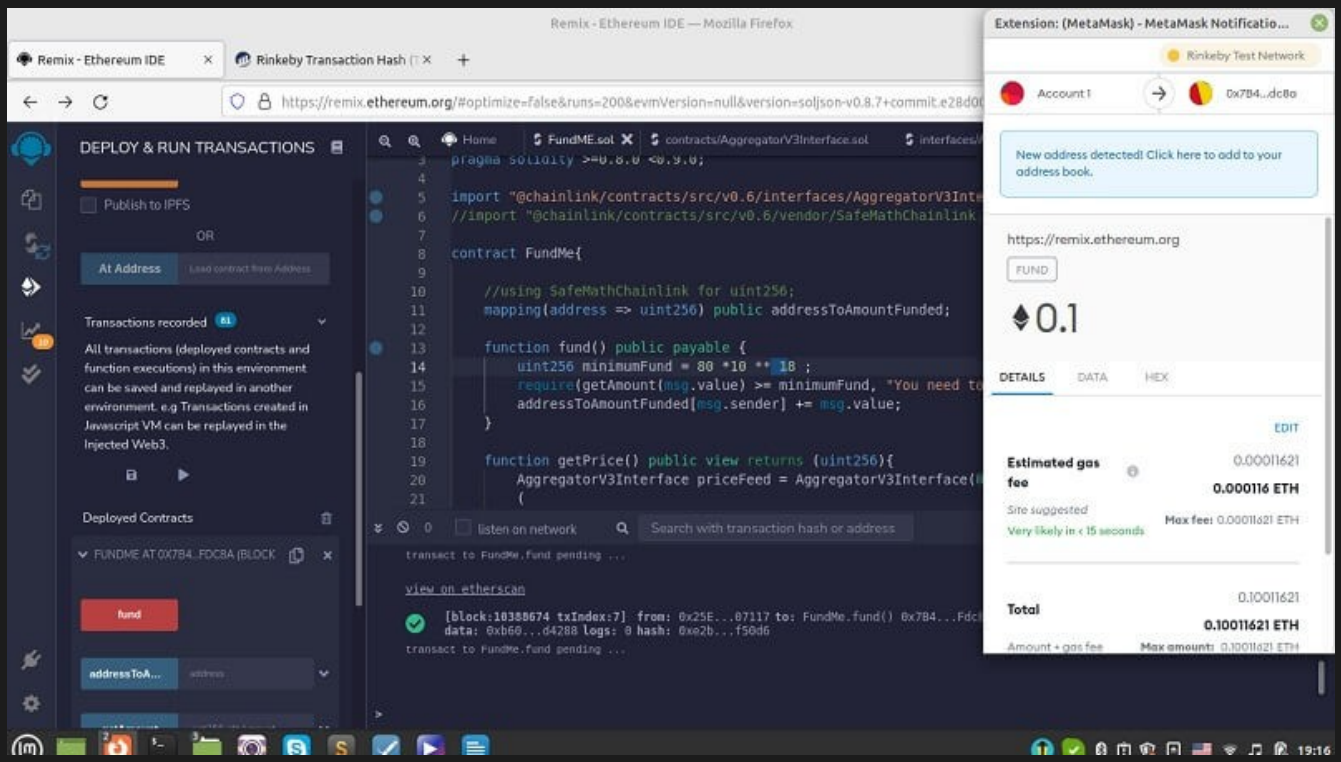
Now, suppose we want the received amount to be at least a certain number. And if the sender sent a lower amount, the transaction gets reverted with a message. To do that, in the fund function, we write:

```
function fund() public payable {
    uint256 minimumFund = 80 * 10 ** 18;
    require (getAmount(msg.value) <= minimumFund, "You need to spend
more

Ether!");
    addressToAmountFunded[msg.sender] += msg.value;
    }
```

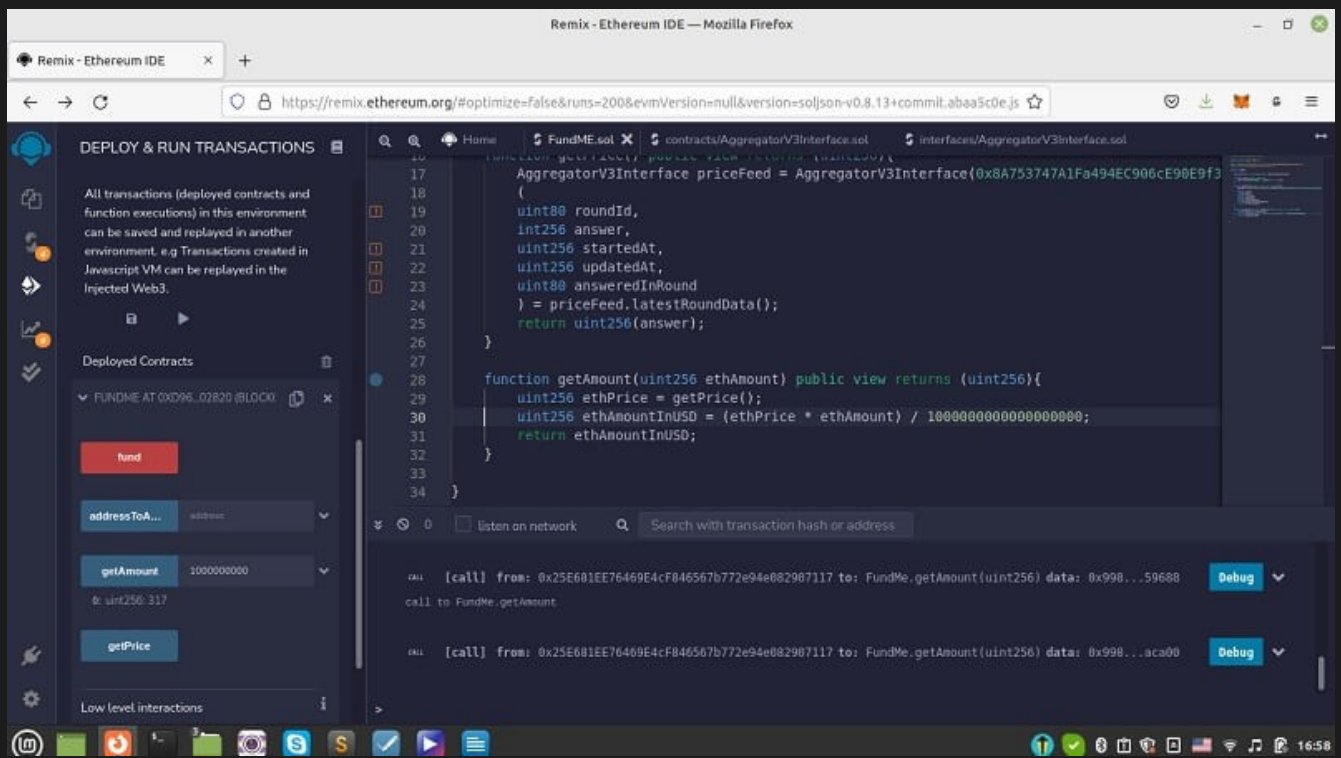
The require function checks for a certain condition for the transaction and if that condition is not met, it will revert it with a message we have determined. Here we have defined the minimum amount to be more than 80 dollars, then multiplied it by 10 to the power of 18 because the default unit is Wei, and then using the getAmount function that we had defined in the first tutorial, converted USD to Ether and compared it with the minimumFund using the require() which is a built-in function of Solidity. If the condition is not met, the sender transaction will be reverted with the message "You need to spend more Ether!".
We can test this function by deploying it and entering 10 in the value box with Wei unit which is a far less amount than 80 dollars and also an amount like 10000000 Gwei which should exceed 80 dollars (it totally depends on the price of Ethereum at any certain time). For the first try, if we enter 10 Wei, the following command will appear with the message with defined



Also if we enter 100000000000000000 Wei, you will face the Metamask popup asking for confirmation.



you can also track both of the tries on Etherscan. The first one shows that the transaction has been reverted and the second one shows the opposite.
Now, we can copy our Rinkeby address and enter it in the address box to check the amount that has been sent.



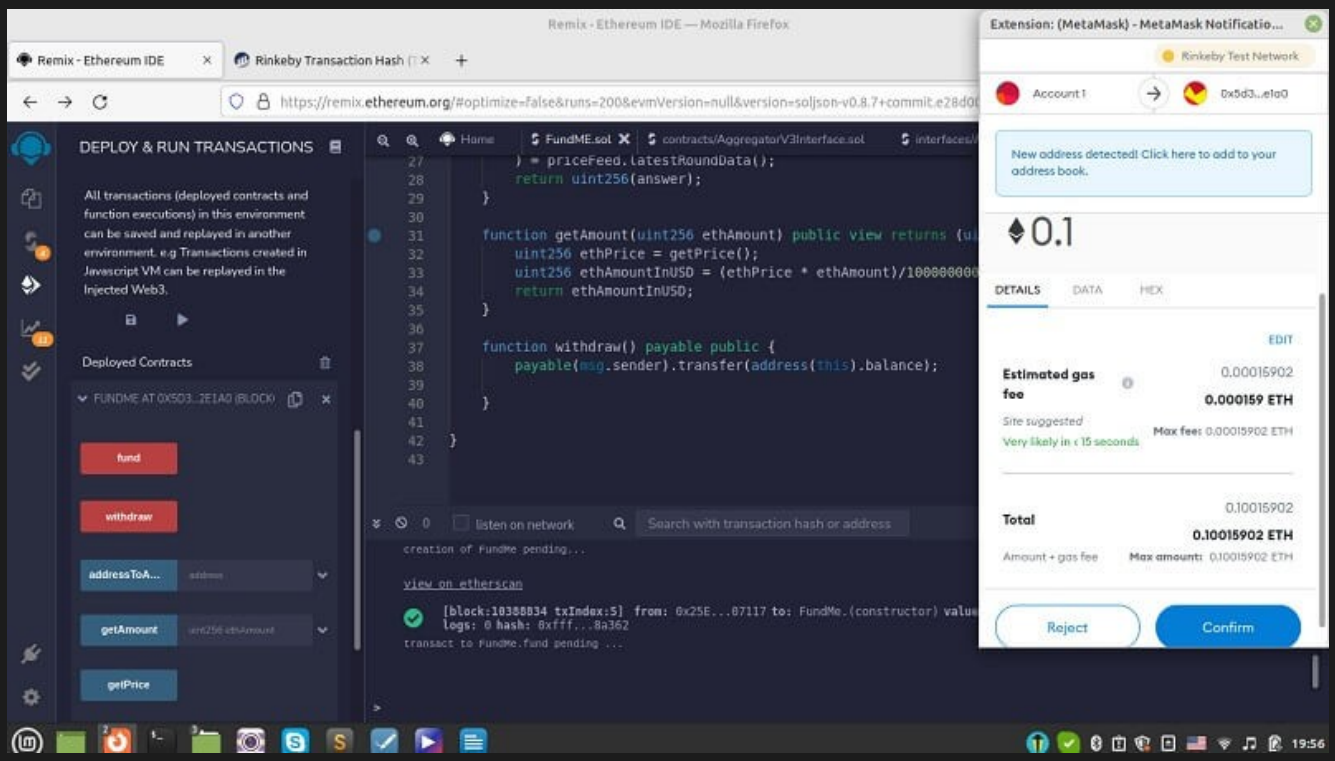
And here you might wonder why the output is 2000000000000000000? The reason here is that we did it twice for the test behind the scenes and because we've sent the fund twice. The output shows the sum of the received value.

WITHDRAW THE FUNDS

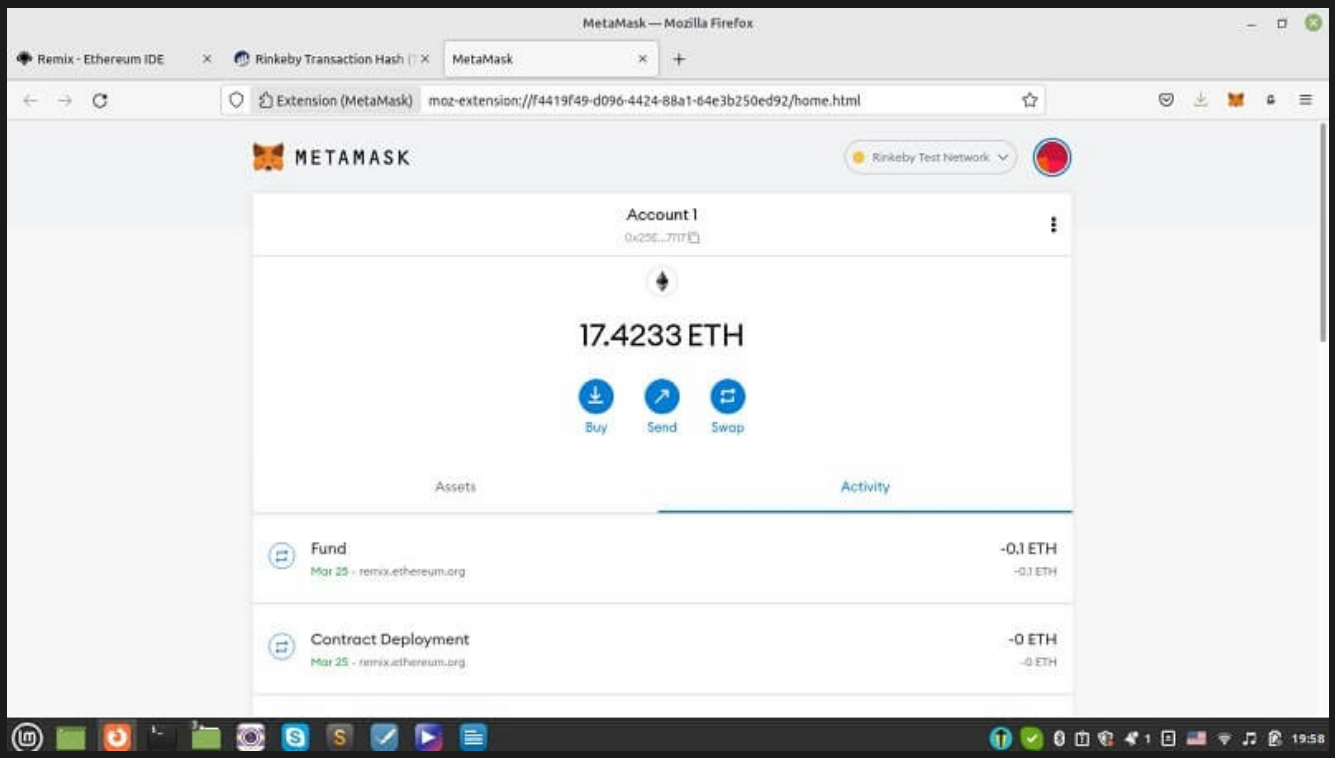
Now, what if we want to get this fund back? There should be a way to withdraw the money. Making another payable function and using .transfer will serve our purpose.

```
function withdraw() payable public {
    payable(msg.sender).transfer(address(this).balance);
    }
```

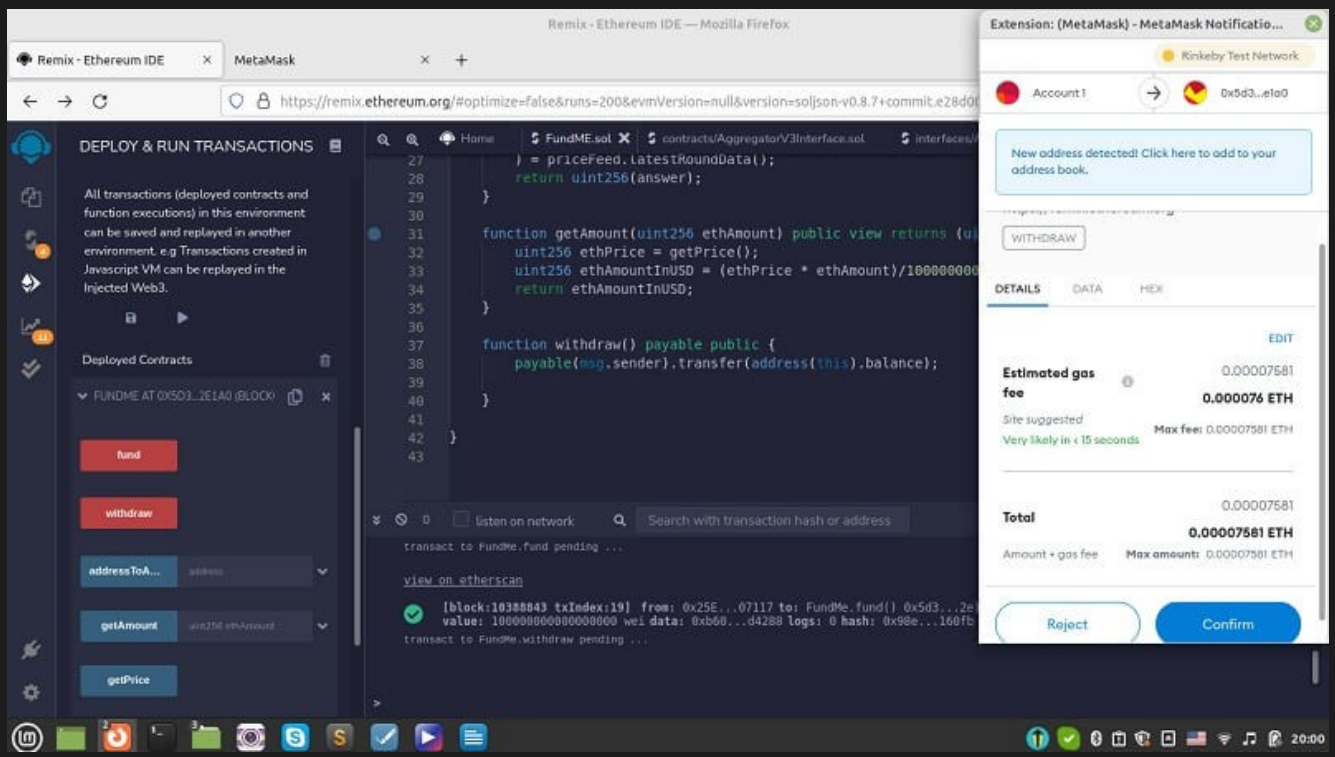
In the above script, notice that we should declare msg.sender as payable, and in general for .send and .transfer, we should declare the address as payable. The attribute this inside address points to the contract, In other words, this here means this contract, and address (this) means the address of this contract.
Once we deploy our contract and send some funds from the Rinkeby Metamask account (0.1 Ether for instance), we will be able to see the Metamask pops up asking for the transaction confirmation:



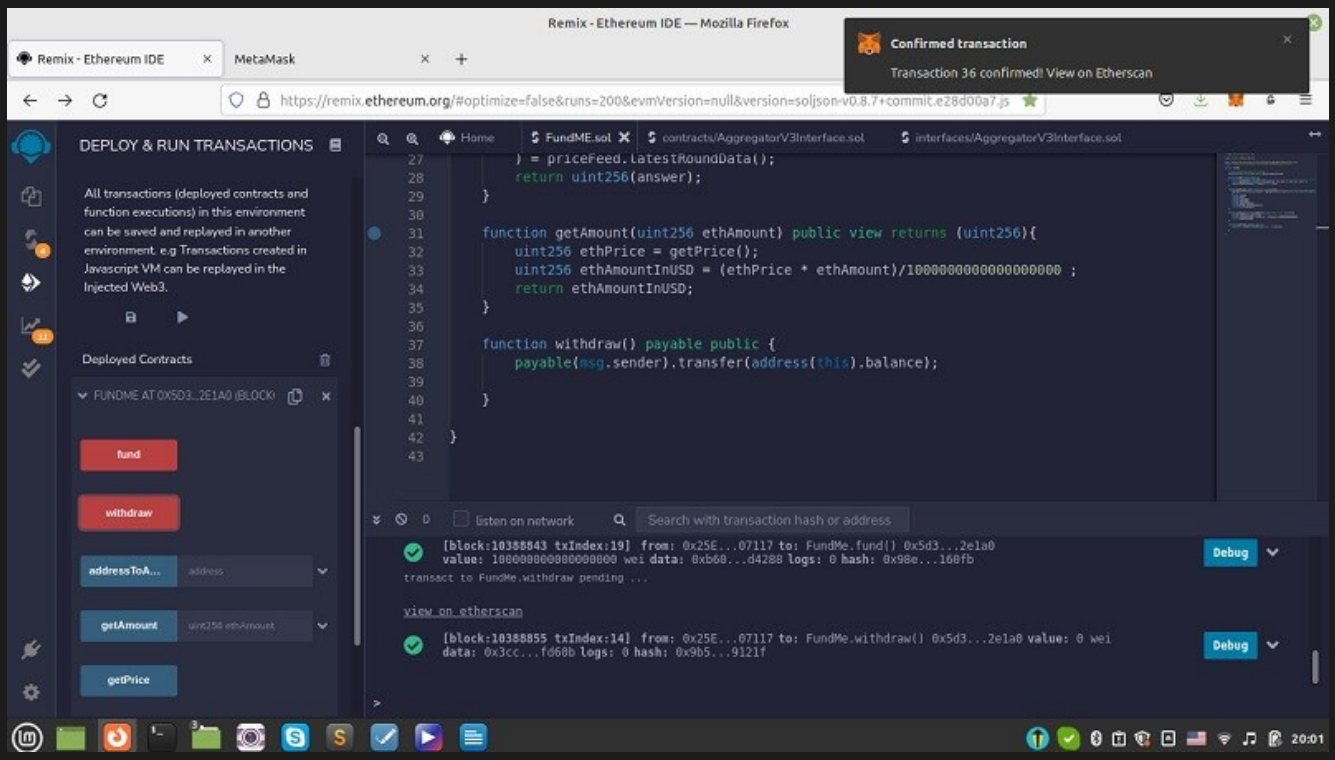
After we confirm the transaction and check our Rinkeby account, we will see that the balance has decreased to 0.1 ETH.



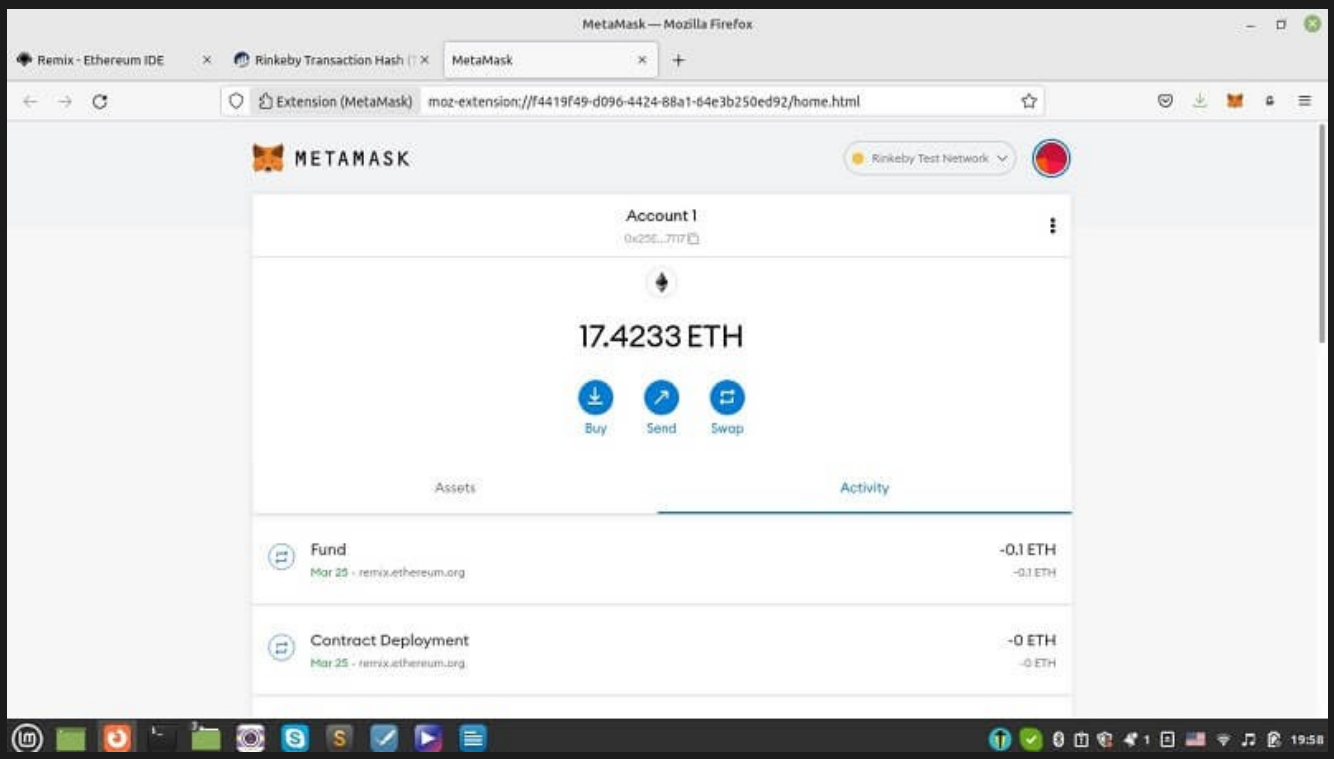
To withdraw the money back from the contract to the account, we press withdraw button and confirm the Metamask asking for the withdrawal confirmation:



After a few seconds, we will be able to see that the transaction has been confirmed:



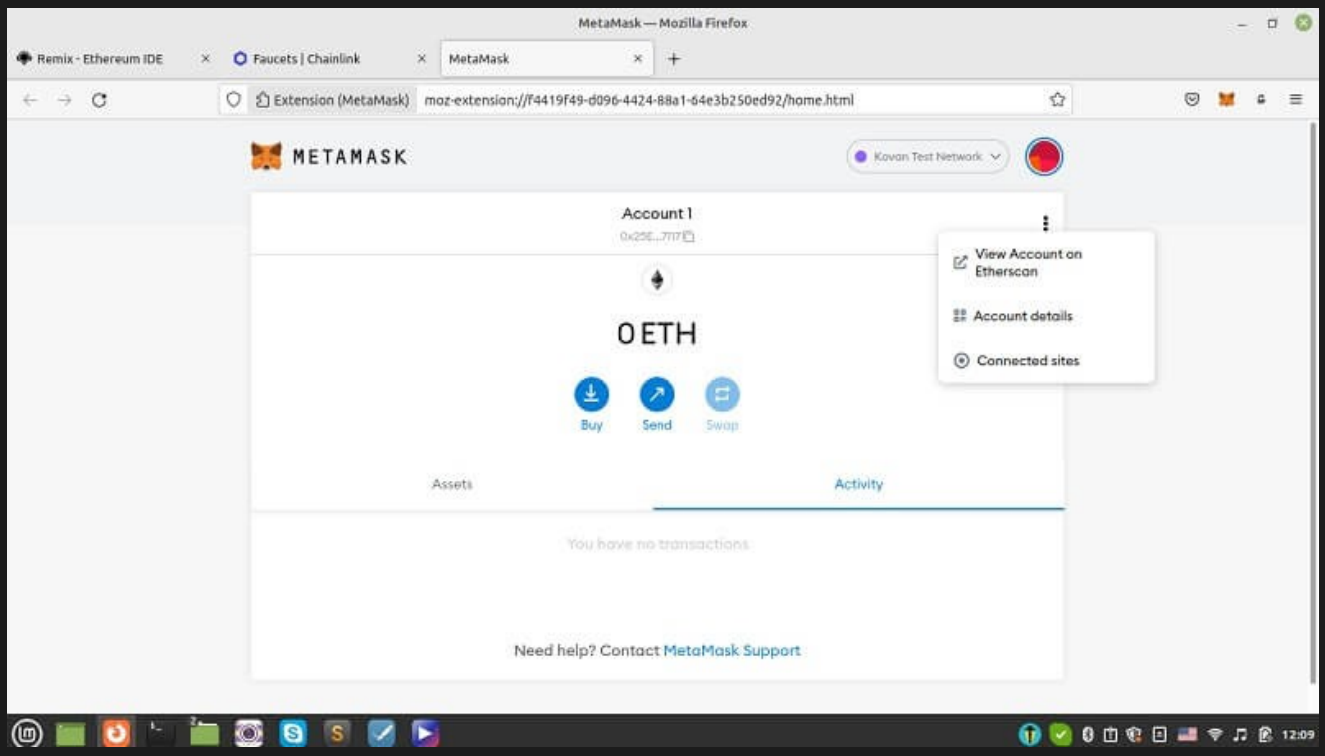
And if we check the account, we will be able to see that 0.1 Ether has returned to the Metamask wallet Rinkeby account.



Hope you have enjoyed using Solidity language for funding in Blockchain.
Now we are going to see how we can give the withdrawal access to the admin so that the withdrawal access won't be given to everyone contributing to the contract but only to the admin of the contract.

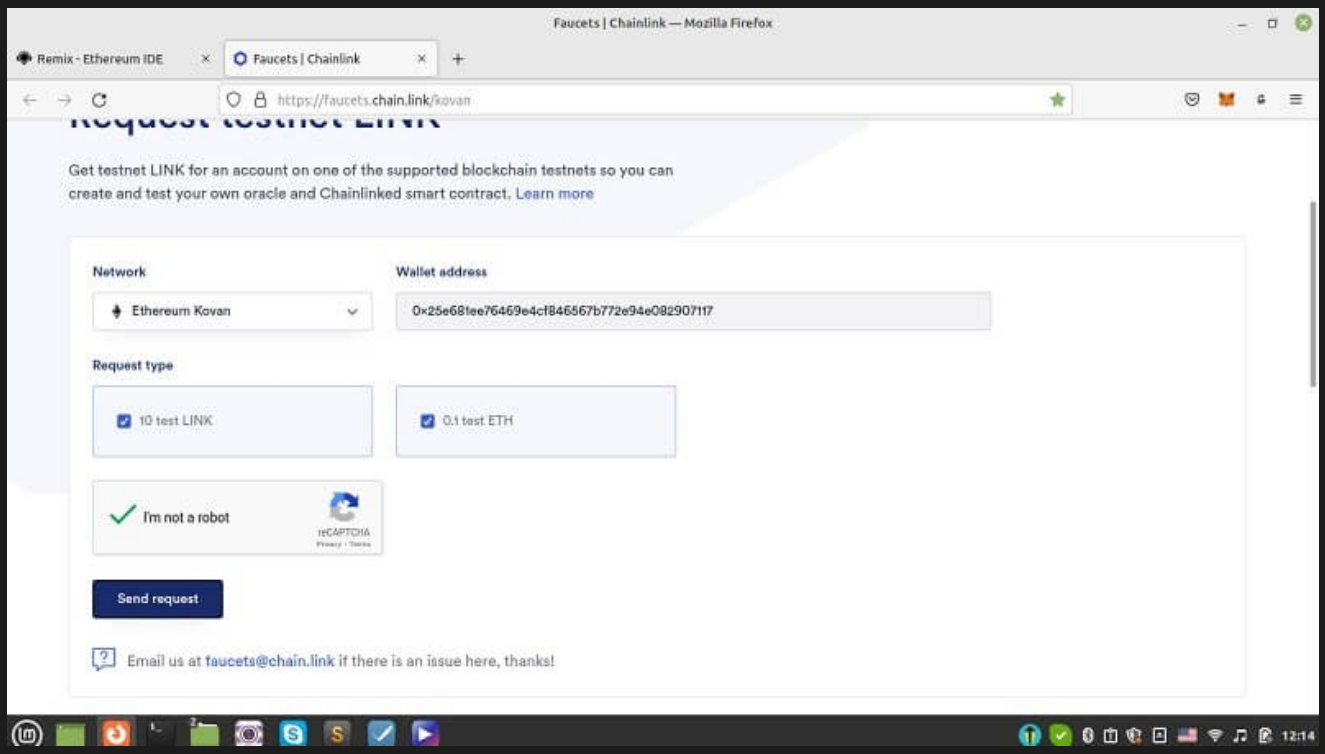
GIVING WITHDRAWAL ACCESS TO ADMIN USING SOLIDITY FOR FUNDING CONTRACTS IN BLOCKCHAIN

Let's see how we can get some test Kovan Ether for our second account. Subsequently, to test if the withdrawal authority only works for the admin of the contract or others can also withdraw the funds. We are also going to get familiar with concepts like constructors and modifiers in Solidity for funding in Blockchain.
In detail, we are going to control the access of the funders and give the withdrawal authority to only one of them. But, before writing the code, we shouldn't forget that in order to test our code, we need another test account with some Ethers in it. So, to get some Ether in the Kovan test net in our Metamask wallet, we can head over to this link and connect our wallet to the website. To do that, click on the connected sites on the hover menu on the top right of the wallet and after that click on manually connect to this site.

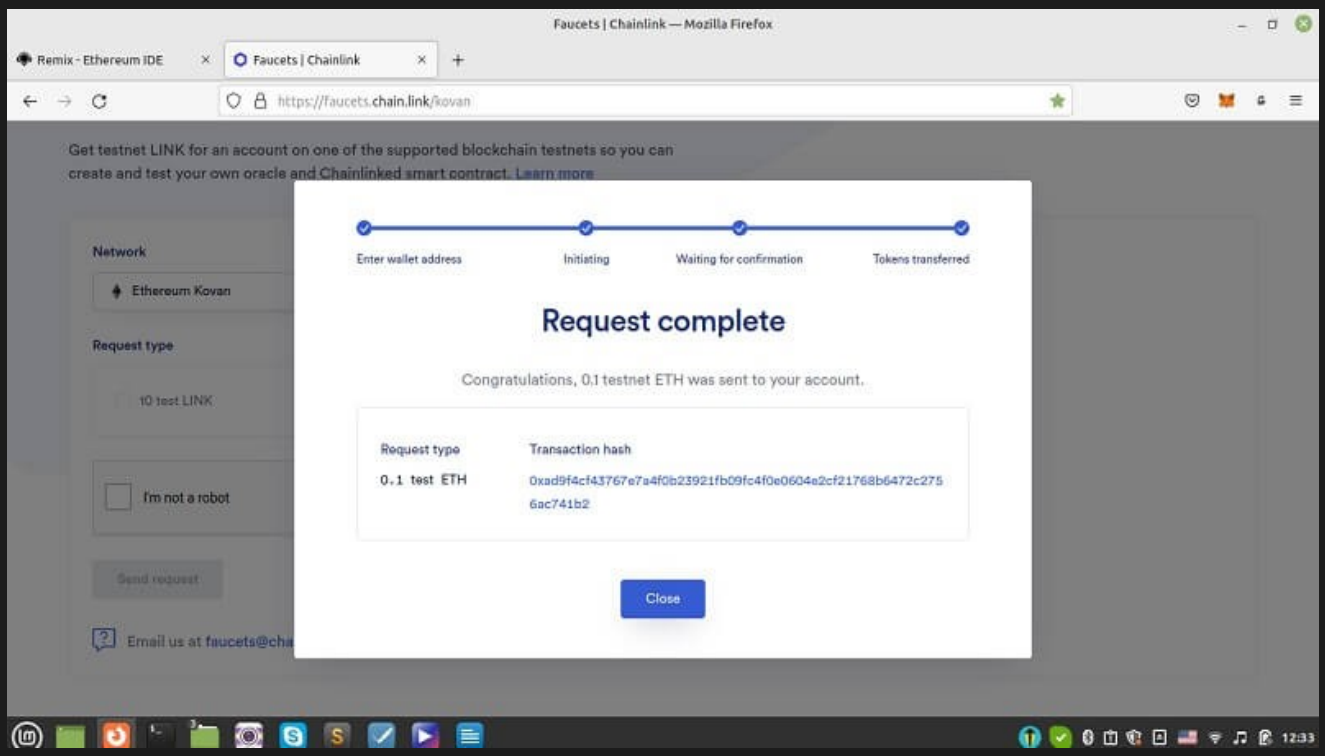


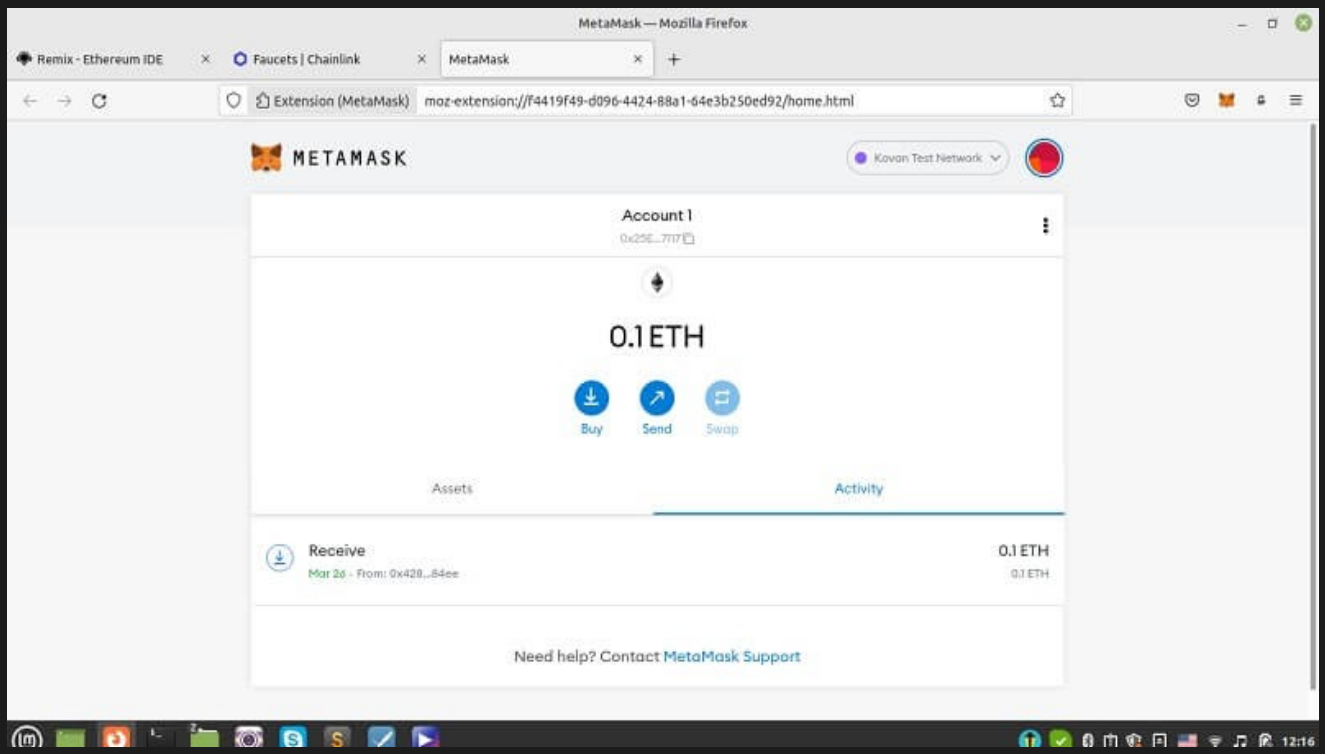
Notice that you cannot connect to the faucet website in another tab. The photo above is just meant to show you where connected sites are located on Metamask. Besides, remember that you should switch to the Kovan test network.

After you have connected your wallet to the chainlink faucet website, press the send request button:



And after a few seconds, you will be able to see that you have successfully received some Kovan Ether in your wallet.

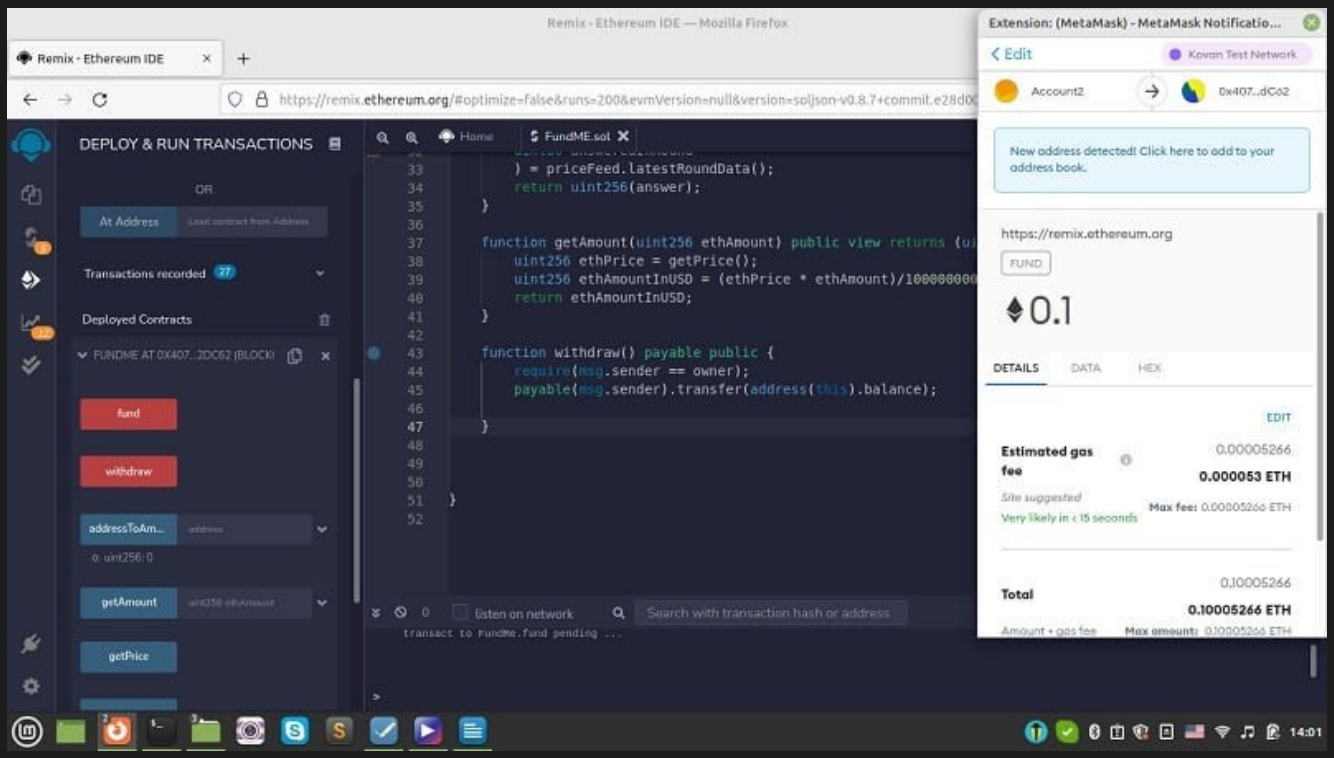




You can do this multiple times to get more Ether, but for this testing, 0.2 ETH is enough. Also, remember that you should open another account for our test and in that account, you can either get some Kovan or Rinkeby Ether.
Now in order to determine the admin of the contract, we add the following codes to the rest of our script inside the contract brackets.

```
address public owner;

constructor() public{
    owner = msg.sender;
    }
```

The above code defines the owner as public and determines the owner as of the deployer of the contract. If we deploy this contract and fund it with 0.1 ETH using our first account, and then press the owner button, we will be able to see that the owner is indeed the account that has deployed the contract.



Now, in order to give the authority of withdrawal to the owner (admin) of this contract, we should use the Require() statement in the withdraw function.

```
function withdraw() payable public {
    require(msg.sender == owner);
    payable(msg.sender).transfer(address(this).balance);
        }
```

Using the above script, we make a condition that if the sender is the owner, they can withdraw from our contract. Otherwise, they cannot do that. If you deploy the contract with the first account, you will be able to see that the owner is the address of the first account. And the same will be true if you deploy it with the 2nd account. Now, if we fund the contract with the 2 accounts and try to withdraw money for both of them, you will see that the withdrawal is only accepted for the account that is the owner of the contract.

Now as you can see, we have defined our modifier to check if the sender is the owner. Then, for our withdraw function, we have declared it with the name of the modifier; "onlyOwner".
If we update our contract with the above scripts, we will be able to see that the same functionality applies to our updated version. We can now test our funding, the conversion of the currency, the address of the funders, the owner of the contract, and the withdrawal authority of the contract.
Congratulation! We have now applied a contract using Solidity for funding in Blockchain. However, it is a minimalistic one, it still has the very basic foundation of every crowdfunding contract inside.

WRAPPING UP

In this article, we focused on funding smart contracts and writing the fundme.sol script in Solidity to receive funds from different accounts. Furthermore, we have got familiar with aggregatorV3Interface as a tool from Chainlink oracle to give us price data feed and then used this data to convert the received funds from Wei, Gwei, and ETH to US dollars.
Moreover, we have managed to cover the overflow pitfall that most programmers face when dealing with big numbers in the Wei unit by using ChinLink safeMath. We have also updated the fundme.sol contract to give the admin the capability of withdrawing funds from the contract account.
And finally, we have managed to complete the funding contract in Solidity. To test whether the withdrawal authority is only for the admin of the contract or others can withdraw as well, we have created another account with some Kovan test Ether in it.
In the end, we saw how constructors and modifiers work in Solidity.