< Blockchain meets AI: The Trust Engine> - 2025.7.2 Week 2
A hands-on guide to develop blockchain apps
(한글버전은 14페이지부터. The Korean version is on page 14)

Written by Jason (No LLM)

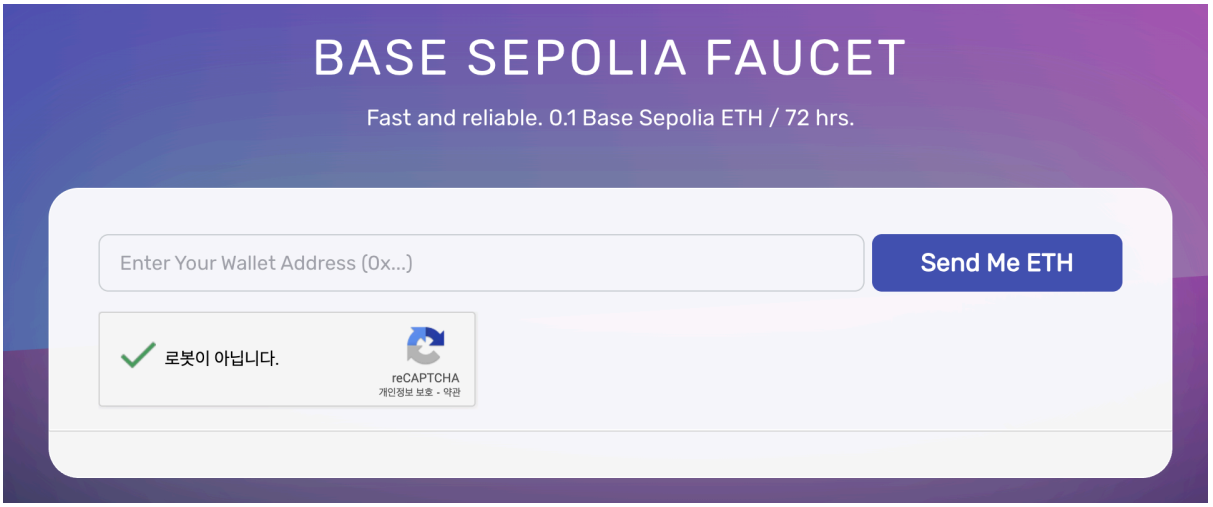
1. Setting up a blockchain dev environment

So far, you've had the experience of receiving, sending, and swapping ETH and tokens on the Base chain. These are all real transactions, and they stay on the Base chain forever. This production environment is called the mainnet, where you have to pay for gas fees with the native token (ETH) for each transaction. On the other hand, the blockchain used for service development is called Testnet. It works exactly like the mainnet, but it has a separate native token for the testnet, which is free to obtain. So developers don't have to worry about paying gas fees to use the testnet. However, testnets can be shut down at any time, so there is no guarantee that data will be stored permanently. Therefore, it is common to use testnets when developing blockchain services.

Each chain has its own testnet. The testnet for the base chain is the Sepolia Base chain. To develop on this testnet, you must first add the testnet to your metamask wallet. Add a network in metamask based on the information below: (Add a network manually)

- Network Name: Base Sepolia Testnet
- RPC URL: https://sepolia.base.org
- Chain ID: 84532
- Currency Symbol: ETH
- Block Explorer: https://sepolia.basescan.org

The next step is to get ETH to use the testnet. All testnets offer the Faucet service, which gives developers free tokens for testing. There are several sites that offer faucet services, but we recommend Alchemy Faucet because it doesn't require a login. Enter your address into the Faucet service and you'll receive free test ETHs. (it takes a little while to receive) Let's check the status of your address in Basescan for Testnet.



2. Developing smart contracts with the Solidity programming language

Programs that run on a blockchain are called smart contracts. A smart contract is a “contract that is represented in digital form, automatically executed, and enforced,” a concept first proposed by Nick Szabo in 1994. Once a record is made on a blockchain, it is stored forever and cannot be tampered with. The same is true for programs. Once deployed on the blockchain, they are executed forever and cannot be modified. Thus, they are similar to legal contracts in the real world. Smart contracts have a number of different characteristics from traditional programs.

- Cannot be modified/tampered with.
    - It is impossible to update the code itself. Therefore, if you want to change the program, you need to deploy a new smart contract instead of updating the existing smart contract.
    - As long as the blockchain platform exists, the program exists forever.
- It runs automatically and without permission.
    - Anyone can access contracts, and if the conditions are met, contracts are automatically executed.
    - Sending a transaction is what triggers a contract.
- The code is transparent and public.
    - The smart contract code is publicly available and the execution history is transparent. (Block Explorer)
- The blockchain executes the contract without a central point. (Decentralization)

    - No separate server or cloud is required to execute contracts. The entire blockchain network executes contracts.
- Secure but also vulnerable
    - Blockchain's security makes it highly secure (unforgeable), but a bug in the code can cause irreparable damage. (Unmodifiable)
    - Because smart contract code is public, it's easy for hackers to exploit vulnerabilities.

Solidity is the most popular programming language for developing smart contracts on Ethereum-like blockchains. It is similar to JavaScript. You can teach yourself the basics of Solidity at these sites

- 2023 Web3@KAIST lectures
- Crypto Zombie
- Speedrun Ethereum
- LearnWeb3
- Solidity Literacy (Korean)

3. Getting started programming Solidity with Remix

The best place to start programming in Solidity is with Remix.

- Remix https://remix.ethereum.org/

The Remix IDE is a web-based integrated development environment (IDE) that helps you develop, deploy, and test Solidity smart contracts. It is a popular tool for beginners and experienced developers alike because it works directly in the browser without installation. The most important feature is that it has a built-in blockchain virtual environment, so you can develop, run, and test without using a real blockchain. Of course, Remix also provides an environment to connect with a wallet to deploy and run on a real blockchain.

Let's write our first Solidity program using Remix. Go to File Explorer > Create new file and create a file called SimpleStorage.sol and copy and paste the code below. This code is a simple contract that calls the setValue() function to change the value of the value variable. However, since the value is stored on

the blockchain, we need to make a transaction to execute it, which will change the state of the blockchain.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.30;

/// @title SimpleStorage
/// @notice A basic contract to store and retrieve a number
contract SimpleStorage {
   // Private variable to store the value
   uint256 private value;

   /// @notice Sets a new value
   /// @param _value The number to store
   function setValue(uint256 _value) public {
       value = _value;
   }

   /// @notice Gets the currently stored value
   /// @return The current value stored in the contract
   function getValue() public view returns (uint256) {
       return value;
   }
}
```

Once you've written your code, compile it on the Solidity compiler tab, making sure that the compiler version on the tab matches the compiler version you specified in your code. Once you have successfully compiled, you can now go to the Deploy & run transactions tab to execute. Here we will deploy and run the contract in the Remix VM environment.

4. Creating an ERC-20 standard token in Remix

This time, let's create a meaningful contract. In Ethereum, tokens mostly follow a standard called ERC-20. In Ethereum, proposing a new technical standard is called an EIP (Ethereum Improvement Proposal), and the smart contract interface standard is called an ERC (Ethereum Request for

Comments). Among them, ERC-20 is the token (Fungible Token) interface standard.

- EIP(Ethereum Improvement Proposal) https://eips.ethereum.org/
- ERC (Ethereum Request for Comments) https://eips.ethereum.org/erc
- ERC-20 https://eips.ethereum.org/EIPS/eip-20

By developing your contract to the ERC-20 standard, you can create tokens that are recognized by wallets and exchanges. You can do this by implementing functions to match the ERC-20 interface below.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

/// @title ERC-20 Token Standard Interface
interface IERC20 {
   // Returns the total token supply
   function totalSupply() external view returns (uint256);

   // Returns the account balance of another account with address 'account'
   function balanceOf(address account) external view returns (uint256);

   // Transfers amount of tokens to recipient
   function transfer(address recipient, uint256 amount) external returns (bool);

   // Returns the amount which spender is still allowed to withdraw from owner
   function allowance(address owner, address spender) external view returns
(uint256);

   // Allows spender to withdraw from your account multiple times, up to amount
   function approve(address spender, uint256 amount) external returns (bool);

   // Transfers tokens from sender to recipient using the allowance mechanism
   function transferFrom(address sender, address recipient, uint256 amount)
external returns (bool);

   // Triggered when tokens are transferred
   event Transfer(address indexed from, address indexed to, uint256 value);

   // Triggered when approve() is called
   event Approval(address indexed owner, address indexed spender, uint256 value);
}
```

In fact, if you can develop an ERC-20 token on your own, you can do most Solidity programming. Beyond that, it just complicates your business logic. Below is the code to implement an ERC-20 token. Copy it, compile it in Remix, and try it out.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.30;

/// @title MyToken - Custom ERC20 Token with Mint and Burn by Owner
contract MyToken {
    string public name;
    string public symbol;
    uint8 public constant decimals = 18;
    uint256 public totalSupply;
    uint256 public maxSupply;
    address public owner;

    mapping(address => uint256) private balances;
    mapping(address => mapping(address => uint256)) private allowances;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    modifier onlyOwner() {
        require(msg.sender == owner, "Not the contract owner");
        _;
    }

    constructor(string memory _name, string memory _symbol, uint256 _maxSupply) {
        name = _name;
        symbol = _symbol;
        maxSupply = _maxSupply;
        owner = msg.sender;
    }

    function balanceOf(address account) public view returns (uint256) {
        return balances[account];
    }

    function transfer(address to, uint256 amount) public returns (bool) {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        _transfer(msg.sender, to, amount);
        return true;
```

```
    }

    function approve(address spender, uint256 amount) public returns (bool) {
        allowances[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function allowance(address _owner, address spender) public view returns
(uint256) {
        return allowances[_owner][spender];
    }

    function transferFrom(address from, address to, uint256 amount) public returns
(bool) {
        require(balances[from] >= amount, "Insufficient balance");
        require(allowances[from][msg.sender] >= amount, "Allowance exceeded");

        allowances[from][msg.sender] -= amount;
        _transfer(from, to, amount);
        return true;
    }

    function _transfer(address from, address to, uint256 amount) internal {
        require(to != address(0), "Invalid recipient");

        balances[from] -= amount;
        balances[to] += amount;
        emit Transfer(from, to, amount);
    }

    /// @notice Mint new tokens to a specified address (owner only)
    function mint(address to, uint256 amount) public onlyOwner {
        require(totalSupply + amount <= maxSupply, "Exceeds max supply");

        balances[to] += amount;
        totalSupply += amount;
        emit Transfer(address(0), to, amount);
    }

    /// @notice Burn tokens from a specified address (owner only)
    function burn(address from, uint256 amount) public onlyOwner {
        require(balances[from] >= amount, "Insufficient balance to burn");

        balances[from] -= amount;
```

```
        totalSupply -= amount;
        emit Transfer(from, address(0), amount);
    }
}
```

5. Implementing ERC-20 tokens by inheriting the Openzeppelin library

So far, we've discussed how to implement an ERC-20 token by hand, but it's actually quite simple to do so by inheriting and creating an existing, well-written library. OpenZeppelin is the most widely used open-source library for Ethereum and smart contract development. It plays a key role in making it easy to implement standard tokens, especially ERC-20 and ERC-721 (NFTs), and avoid security vulnerabilities.

- OpenZeppelin Docs https://docs.openzeppelin.com/
- OpenZeppelin Github https://github.com/OpenZeppelin/openzeppelin-contracts

This is an example of implementing a token by inheriting the OpenZeppelin ERC20 contract. It takes only 5 lines of code to create a new token.

```
// contracts/GLDToken.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract GLDToken is ERC20 {
    constructor(uint256 initialSupply) ERC20("Gold", "GLD") {
        _mint(msg.sender, initialSupply);
    }
}
```

We've already seen that many contracts are implemented based on the OpenZeppelin library and run reliably, so it's convenient and safe to develop smart contracts based on it whenever possible.

6. Implementing Stablecoins, an Application of ERC-20

ERC-20 is a standard for creating tokens. The ERC-20 token standard allows you to create tokens with a variety of functions.

- Utility Tokens (e.g. $LINK)
- Governance Tokens (e.g. $UNI)
- Stablecoins (e.g. $USDT)
- Memecoins (e.g. $PEPE)
- Wrapped tokens (e.g. $WETH)

If you're developing a hot stablecoin, how do you implement it? It's simple. All you need is the ability to mint and burn the underlying ERC-20 token. If someone wants stablecoins, they send you fiat currency like USD, you mint the same amount of stablecoins, and if they want their fiat back, you burn the stablecoins and send their fiat back. But doesn't this sound strange? How can you send fiat currency to a blockchain? It's impossible. You need a separate service to receive, store, and return the fiat, which means that the existence of the physical asset must be guaranteed off-chain. That's why stablecoins are composed of two parts: an off-chain part that manages the fiat and an on-chain part that circulates stablecoins on the blockchain, and the on-chain part can be implemented as an ERC-20 token. Below is an example of a contract implementing a stablecoin.

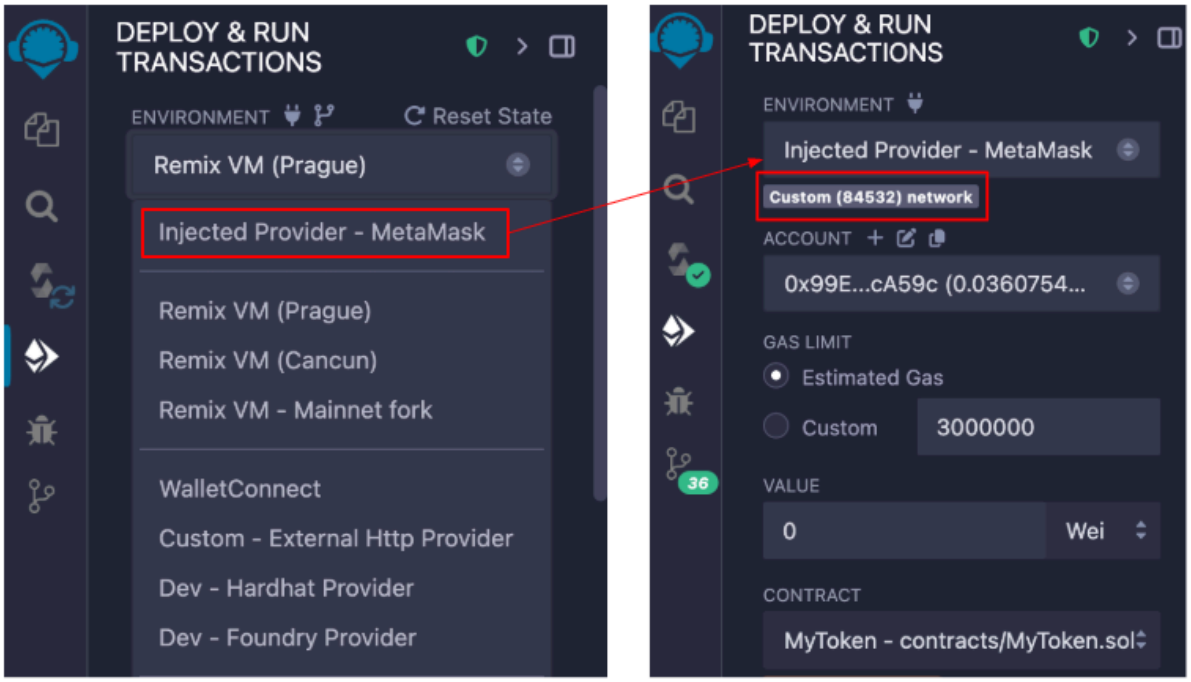
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

/// @title MyStablecoin - ERC-20 Stablecoin Example
contract MyStablecoin is ERC20, Ownable {
    constructor(string memory name, string memory symbol)
        ERC20(name, symbol)
        Ownable(msg.sender)
    {}

    /// @notice Mint tokens to an address (onlyOwner)
    function mint(address to, uint256 amount) external onlyOwner {
        require(to != address(0), "Cannot mint to zero address");
        _mint(to, amount);
    }
```

```
    /// @notice Burn tokens from an address (onlyOwner)
    function burn(address from, uint256 amount) external onlyOwner {
        require(from != address(0), "Cannot burn from zero address");
        _burn(from, amount);
    }
}
```

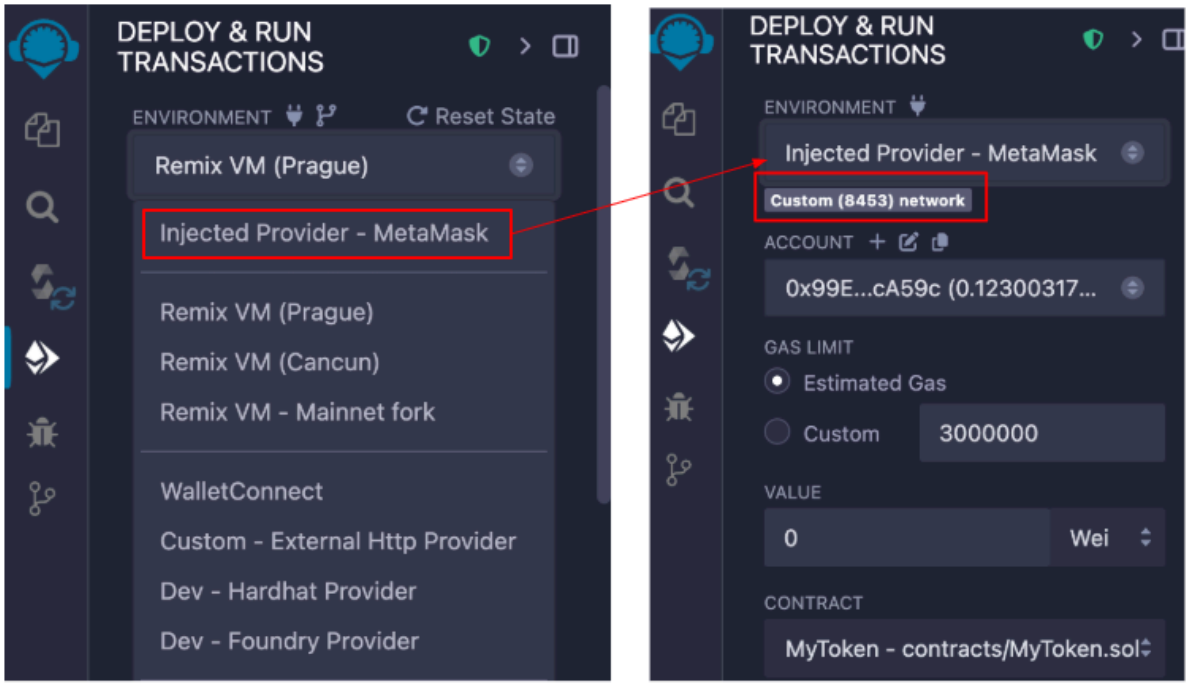
USDT, the most circulating stablecoin in the real world, is also an ERC-20 token. The contract code is simple, and it's amazing that such a simple code is now worth $157B. However, USDT is not only issued on Ethereum, but on dozens of blockchains, so the implementation on each chain can be slightly different.

7. Deploy and try out the ERC-20 token you implemented on the Base Sepolia testnet

You can mint the token you implemented in section 5 on a real blockchain instead of the Remix virtual environment. In this example, let's mint your ERC-20 token on the Base Sepolia testnet. First, log in to Metamask, select the Base Sepolia testnet from the network and choose the address you want to deploy the contract to, this address will be the owner of the contract. Next, in the Remix Deploy & Run tab, select Environment as ‘Injected Provider - Metamask’ and you should be able to connect with Metamask and see your address in Account. If you don't see your Metamask address here, it's not connected to your wallet, and make sure the Chain ID is the one you want. Now, in Deploy, you can set the input variables and press the transact button to deploy.



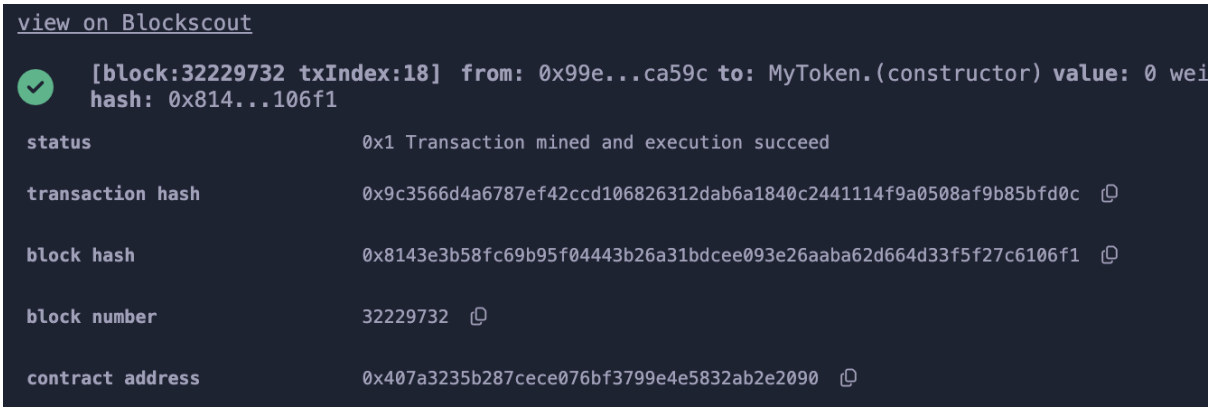
If you connected to the Base Sepolia Testnet (Chain ID = 84532)



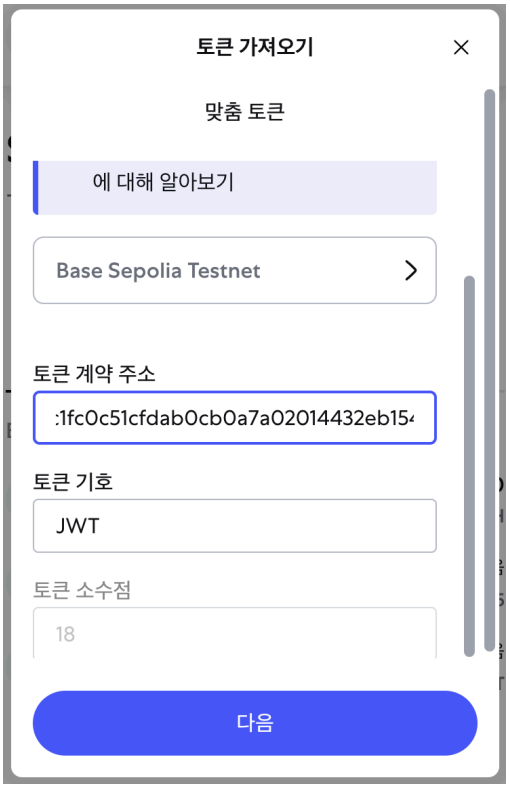
If you connected to the Base Mainnet (Chain ID = 8453)

Now that we're in a real blockchain environment, we need to pay for gas. If Remix warns you about the gas fee, just run it anyway, and Metamask will set it to the appropriate gas fee and send the transaction to the chain. After a final

confirmation from Metamask, you can see in the Remix console window that the contract creation transaction has been sent to the chain. If you expand the log as shown below, you can copy the contract address. If you look up this address in Sepolia Basescan, you can see that the contract has been successfully deployed.



Now you have a real token, deployed on a real blockchain. Call the mint() function to mint some tokens for you. The beauty of Remix is that you can call any function in Remix without developing a separate frontend. Run a Basescan to verify that the mint() function ran successfully and that your tokens are now in your wallet. To do this, you'll first need to get Metamask to recognize the tokens you created.



Afterward, you check your wallet to see if the tokens have arrived. But is the number of tokens less than you thought? This is where the concept of decimal comes in. In Solidity, we don't have floating point (float, double), so all numbers are stored and calculated in integer units (uint). So Solidity introduced decimals to represent decimal units as integers. If decimals = 18, it would represent the number of tokens as below.

- 1.0 tokens = 1 * 10^18 = 1000000000000000000 (integer)
- 0.5 tokens = 0.5 * 10^18 = 500000000000000000

So, when you send 1 token, you need to put 1000000000000000000 as the input. This wouldn't be an issue when testing only within Remix (we actually send and receive tokens in decimal units), but Metamask is calculating and displaying decimals. Most tokens will specify 18 because Ethereum uses 18 for decimals. Of course, you can specify any other value. It's complicated to calculate every time you type it, so a good way to reduce errors is to keep the Unit Converter page open and copy and paste the value you want to enter (Ether) with decimals (Wei).

- 10 Ether = 10000000000000000000 Wei

Once your tokens have been successfully distributed to the chain and you've received them to your address, it's time to send them to someone else. You can do this from your wallet or with the transfer function in Remix. Of course, you'll need to tell them the contract address so that their wallet recognizes your token.

< Blockchain meets AI: The Trust Engine> - 2025.7.2 Week 2
A hands-on guide to develop blockchain apps
(한글버전은 14페이지부터. The Korean version is on page 14)

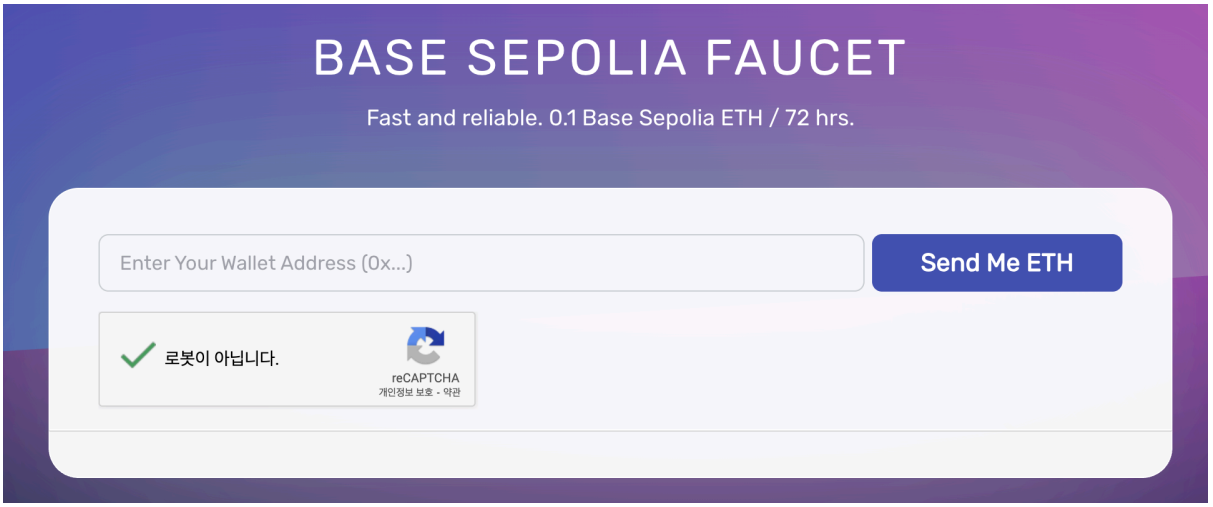
Written by Jason (No LLM)

1. 블록체인 개발 환경 셋업하기
지금까지 당신은 Base chain에서 ETH와 토큰을 받고 보내고 스왑하는 경험을 해 보았습니다. 이것은 모두 실제 트랜잭션이고, Base chain 위에 영원히 남게 됩니다. 이런 Production 환경을 mainnet이라고 합니다. 여기에선 트랜잭션마다 native token(ETH)로 가스비를 지불해야 하고, 영원히 블록체인에 남게 됩니다. 반면에 서비스 개발을 위해 사용하는 블록체인을 Testnet이라고 합니다. 이것은 메인넷과 똑같이 동작하지만, 테스트넷을 위한 별도의 native token을 가지고 있고 이건 무료로 구할 수 있습니다. 따라서 개발자는 테스트넷을 이용하며 지불하는 gas fee를 걱정하지 않아도 됩니다. 다만, 테스트넷은 언제든 중단할 수 있기 때문에 데이터가 영구히 저장되는걸 보장하진 않습니다. 따라서, 블록체인 서비스 개발할 때는 테스트넷을 사용하는게 일반적입니다.

각 체인은 자신의 테스트넷을 가지고 있습니다. Base chain의 테스트넷은 Sepolia Base chain 입니다. 이 테스트넷에서 개발하기 위해선 우선 metamask 지갑에 테스트넷을 추가해야 합니다. 아래의 정보를 바탕으로 metamask에서 네트워크를 추가합니다. (Add network)

- Network Name: Base Sepolia Testnet
- RPC URL: https://sepolia.base.org
- Chain ID: 84532
- Currency Symbol: ETH
- Block Explorer: https://sepolia.basescan.org

다음 스텝은 테스트넷을 이용하기 위한 ETH를 얻는 것 입니다. 모든 테스트넷은 Faucet 서비스를 제공해서 개발자들에게 무료로 테스트용 토큰을 제공하고 있습니다. 여러 사이트에서 Faucet 서비스를 제공하고 있습니다. 그 중에 로그인이 필요없는 Alchemy Faucet을 추천합니다. Faucet 서비스에 당신의 주소를 입력하면 무료 테스트용 ETH를 받을 수 있습니다. (받는데 약간 시간이 걸립니다) 당신 주소의 상태를 테스트넷을 위한 Basescan에서 확인해 봅시다.



2. Solidity 프로그래밍 언어로 스마트 컨트랙트 개발하기

블록체인에서 실행되는 프로그램을 스마트 컨트랙트라고 합니다. 스마트 컨트랙트는 1994년 닉 자보에 의해 처음 제시된 개념으로, '디지털 형식으로 표현되고, 자동으로 실행되며, 강제되는 계약'을 말합니다. 블록체인은 한번 기록이 되면 영원히 저장되며 변조 불가능합니다. 프로그램도 마찬가지입니다. 한번 블록체인에 배포되면 영원히 수정이 불가능한 형태로 실행됩니다. 따라서, 현실 세계의 법적 계약과 유사한거죠. 스마트 컨트랙트는 기존 프로그램과 여러 다른 특징을 가지고 있습니다.

- 수정/변조가 불가능하다.
  - 코드 업데이트 자체가 불가능하다. 따라서 프로그램을 변경하려면 스마트 컨트랙트 대신 새로운 스마트 컨트랙트를 배포해야 한다.
  - 블록체인 플랫폼이 존재하는 한 프로그램은 영원히 존재한다.
- 무허가로 자동 실행된다.
  - 누구나 컨트랙트를 접근할 수 있고, 조건이 충족되면 자동으로 컨트랙트가 실행된다.
  - 트랜잭션을 보내는 것이 컨트랙트를 실행시키는 것이다.
- 코드가 투명하게 공개되어 있다.
  - 스마트 컨트랙트 코드는 누구나 볼 수 있고 실행 내역도 투명하게 확인할 수 있다. (Block Explorer)
- 실행의 주체가 없이 블록체인이 실행한다. (탈중앙화)
  - 컨트랙트를 실행하기 위해 별도의 서버나 클라우드가 필요하지 않다. 블록체인 네트워크 전체가 컨트랙트를 실행한다.
- 보안에 강하면서도 취약할 수 있다.
  - 블록체인이 가진 보안성으로 인해 보안이 강하지만(위변조 불가능) 코드에 버그가 있는 경우 돌이킬 수 없는 피해를 볼 수 있다. (수정 불가능)

    - 스마트 컨트랙트 코드가 공개되어 있기 때문에 해커에게 취약점이 노출되기 쉽다.

이더리움 계열 블록체인에서는 스마트 컨트랙트를 개발하는 프로그래밍 언어로 Solidity를 가장 많이 사용합니다. 자바스크립트와 유사합니다. Solidity 기초는 아래 사이트들에서 스스로 공부할 수 있다.

- 2023 Web3@KAIST lectures
- Crypto Zombie
- Speedrun Ethereum
- LearnWeb3
- Solidity Literacy (Korean)

3. Remix로 **Solidity** 프로그래밍 시작하기

Solidity 프로그래밍을 시작하기 가장 좋은 서비스는 Remix입니다.

- Remix https://remix.ethereum.org/

Remix IDE는 Solidity 스마트 컨트랙트를 개발, 배포, 테스트할 수 있도록 지원하는 웹 기반 통합 개발 환경(IDE)입니다. 설치 없이 브라우저에서 바로 사용 가능하여 초보자와 숙련자 모두에게 널리 쓰이는 도구입니다. 가장 중요한 특징은 블록체인 가상환경을 내장하고 있기 때문에 실제 블록체인을 이용하지 않아도 개발과 실행, 테스트를 해 볼 수 있다는 것입니다. 물론 지갑과 연결해서 실제 블록체인에 배포하고 실행하는 환경도 제공하고 있습니다.

그럼 Remix를 이용해 첫 Solidity 프로그램을 작성해 봅시다. File Explorer > Create new file에서 SimpleStorage.sol 이라는 파일을 만들고 아래 코드를 복사해서 붙여 넣습니다. 이 코드는 setValue() 함수를 호출해서 value 값을 바꾸는 간단한 컨트랙트입니다. 하지만, value 값은 블록체인에 저장되어 있기 때문에 이를 실행하기 위해서는 트랜잭션을 발생시켜야 하고 그때 블록체인의 상태가 변하게 됩니다.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.30;
```

```
/// @title SimpleStorage
/// @notice A basic contract to store and retrieve a number
contract SimpleStorage {
    // Private variable to store the value
    uint256 private value;

    /// @notice Sets a new value
    /// @param _value The number to store
    function setValue(uint256 _value) public {
        value = _value;
    }

    /// @notice Gets the currently stored value
    /// @return The current value stored in the contract
    function getValue() public view returns (uint256) {
        return value;
    }
}
```

코드를 작성했다면, Solidity compiler 탭에서 Compile을 합니다. 이 때 코드에서 지정한 컴파일러 버전과 탭의 컴파일러 버전이 일치해야 합니다. 컴파일 성공했다면 이제 실행하기 위해 Deploy & run transactions 탭으로 이동합니다. 여기에서 Remix VM 환경에서 컨트랙트를 Deploy하고 실행해 봅니다.

4. Remix에서 **ERC-20** 표준 토큰 만들기

이번에는 본격적으로 제대로 된 컨트랙트를 만들어 봅시다. 이더리움에서 토큰들은 대부분 ERC-20이라는 표준을 따릅니다. 이더리움에서는 새로운 기술 표준을 제안하는 것을 EIP(Ethereum Improvement Proposal)이라 하고, 이 중에서 스마트 컨트랙트 인터페이스 표준을 ERC (Ethereum Request for Comments)라고 합니다. 그 중에 ERC-20이 토큰(Fungible Token) 인터페이스 표준입니다.

- EIP(Ethereum Improvement Proposal) https://eips.ethereum.org/
- ERC (Ethereum Request for Comments) https://eips.ethereum.org/erc
- ERC-20 https://eips.ethereum.org/EIPS/eip-20

ERC-20 표준에 맞게 컨트랙트를 개발하면 지갑과 거래소에서 인식되는 토큰을 만들 수 있습니다. 아래 ERC-20 인터페이스에 맞게 함수를 구현하면 됩니다.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

/// @title ERC-20 Token Standard Interface
interface IERC20 {
    // Returns the total token supply
    function totalSupply() external view returns (uint256);

    // Returns the account balance of another account with address 'account'
    function balanceOf(address account) external view returns (uint256);

    // Transfers amount of tokens to recipient
    function transfer(address recipient, uint256 amount) external returns (bool);

    // Returns the amount which spender is still allowed to withdraw from owner
    function allowance(address owner, address spender) external view returns
(uint256);

    // Allows spender to withdraw from your account multiple times, up to amount
    function approve(address spender, uint256 amount) external returns (bool);

    // Transfers tokens from sender to recipient using the allowance mechanism
    function transferFrom(address sender, address recipient, uint256 amount)
external returns (bool);

    // Triggered when tokens are transferred
    event Transfer(address indexed from, address indexed to, uint256 value);

    // Triggered when approve() is called
    event Approval(address indexed owner, address indexed spender, uint256 value);
}
```

사실 ERC-20 토큰을 스스로 개발할 수 있다면 대부분의 Solidity 프로그래밍을 할 수 있습니다. 그 이외에는 비즈니스 로직이 복잡해 질 뿐입니다. 아래는 ERC-20 토큰을 구현한 코드입니다. 복사해서 Remix에서 컴파일하고 실행해 보세요.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.30;

/// @title MyToken - Custom ERC20 Token with Mint and Burn by Owner
```

```
contract MyToken {
    string public name;
    string public symbol;
    uint8 public constant decimals = 18;
    uint256 public totalSupply;
    uint256 public maxSupply;
    address public owner;

    mapping(address => uint256) private balances;
    mapping(address => mapping(address => uint256)) private allowances;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    modifier onlyOwner() {
        require(msg.sender == owner, "Not the contract owner");
        _;
    }

    constructor(string memory _name, string memory _symbol, uint256 _maxSupply) {
        name = _name;
        symbol = _symbol;
        maxSupply = _maxSupply;
        owner = msg.sender;
    }

    function balanceOf(address account) public view returns (uint256) {
        return balances[account];
    }

    function transfer(address to, uint256 amount) public returns (bool) {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        _transfer(msg.sender, to, amount);
        return true;
    }

    function approve(address spender, uint256 amount) public returns (bool) {
        allowances[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function allowance(address _owner, address spender) public view returns
(uint256) {
        return allowances[_owner][spender];
```

```
    }

    function transferFrom(address from, address to, uint256 amount) public returns
(bool) {
        require(balances[from] >= amount, "Insufficient balance");
        require(allowances[from][msg.sender] >= amount, "Allowance exceeded");

        allowances[from][msg.sender] -= amount;
        _transfer(from, to, amount);
        return true;
    }

    function _transfer(address from, address to, uint256 amount) internal {
        require(to != address(0), "Invalid recipient");

        balances[from] -= amount;
        balances[to] += amount;
        emit Transfer(from, to, amount);
    }

    /// @notice Mint new tokens to a specified address (owner only)
    function mint(address to, uint256 amount) public onlyOwner {
        require(totalSupply + amount <= maxSupply, "Exceeds max supply");

        balances[to] += amount;
        totalSupply += amount;
        emit Transfer(address(0), to, amount);
    }

    /// @notice Burn tokens from a specified address (owner only)
    function burn(address from, uint256 amount) public onlyOwner {
        require(balances[from] >= amount, "Insufficient balance to burn");

        balances[from] -= amount;
        totalSupply -= amount;
        emit Transfer(from, address(0), amount);
    }
}
```

5. Openzeppelin 라이브러리 상속 받아 **ERC-20** 토큰 구현하기

지금까지는 ERC-20 토큰을 직접 구현하는 방법에 대해 설명했습니다. 하지만, 실제로는 기존에 잘 작성된 라이브러리를 상속받아서 만들면 아주 간단하게

구현할 수 있습니다. 더군다나 이미 검증된 코드라 오류도 없습니다.
OpenZeppelin은 이더리움과 스마트 컨트랙트 개발을 위한 가장 널리 사용되는 오픈소스 라이브러리입니다. 특히 ERC-20, ERC-721(NFT) 같은 표준 토큰을 쉽게 구현하고, 보안 취약점을 방지하는 데 핵심적인 역할을 합니다.

- OpenZeppelin Docs https://docs.openzeppelin.com/
- OpenZeppelin Github https://github.com/OpenZeppelin/openzeppelin-contracts

OpenZeppelin ERC20 컨트랙트를 상속받아서 토큰을 구현한 예제입니다. 고작 5줄의 코드로 새로운 토큰을 만들어 낼 수 있습니다.

```
// contracts/GLDToken.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract GLDToken is ERC20 {
   constructor(uint256 initialSupply) ERC20("Gold", "GLD") {
       _mint(msg.sender, initialSupply);
   }
}
```

이미 많은 컨트랙트가 OpenZeppelin 라이브러리를 기반으로 구현됐고 안정적으로 실행되는 것을 확인했습니다. 따라서 되도록이면 이 라이브러리를 기반으로 스마트 컨트랙트를 개발하는 것이 편리하고 안전한 방법입니다.

6. ERC-20의 응용**, Stablecoin** 구현하기

ERC-20은 토큰을 만드는 표준입니다. ERC-20 토큰 표준으로 다양한 기능을 하는 토큰을 만들 수 있습니다.

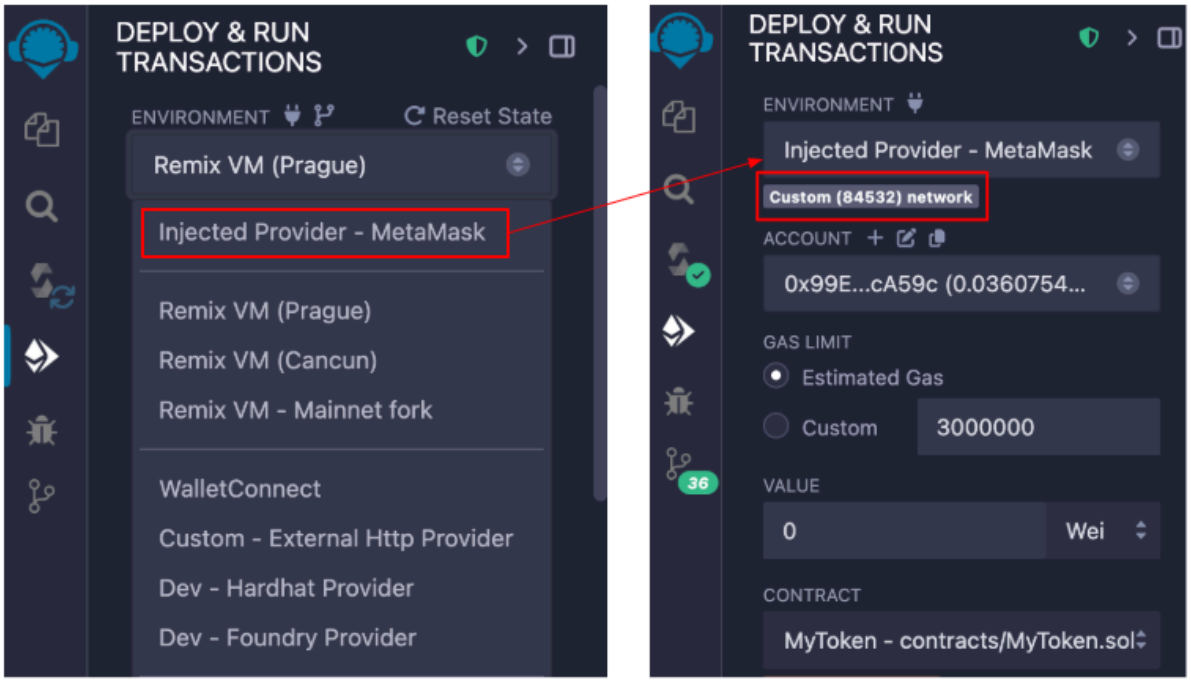
- Utility Tokens (e.g. $LINK)
- Governance Tokens (e.g. $UNI)
- Stablecoins (e.g. $USDT)
- Memecoins (e.g. $PEPE)
- Wrapped tokens (e.g. $WETH)

요즘 핫한 스테이블코인을 개발한다면 어떻게 구현하면 될까요? 간단합니다. 기본 ERC-20 토큰에 발행(mint)하고 소각(burn)하는 기능만 있으면 됩니다. 스테이블코인을 원하는 사람이 법정화폐를 보내면 동일한 만큼의 스테이블코인을 발행(mint)해 주고, 반대로 법정화폐를 돌려 받고 싶으면 스테이블코인을 소각(burn)하고 돌려주면 됩니다. 하지만, 뭔가 이상하지 않나요? 법정화폐를 어떻게 블록체인으로 보낼 수 있을까요? 이것은 불가능합니다. 법정화폐를 받아서 보관하고 반환하는 별도의 서비스가 필요합니다. 즉, 실제 자산의 존재 여부는 오프체인에서 보증해야 합니다. 그래서 스테이블코인은 법정화폐를 관리하는 오프체인 파트와 블록체인에서 유통하는 온체인 파트 두 부분으로 구성되고, 온체인 파트가 ERC-20 토큰으로 구현될 수 있는 것입니다. 아래는 스테이블코인을 구현한 컨트랙트 예시입니다.

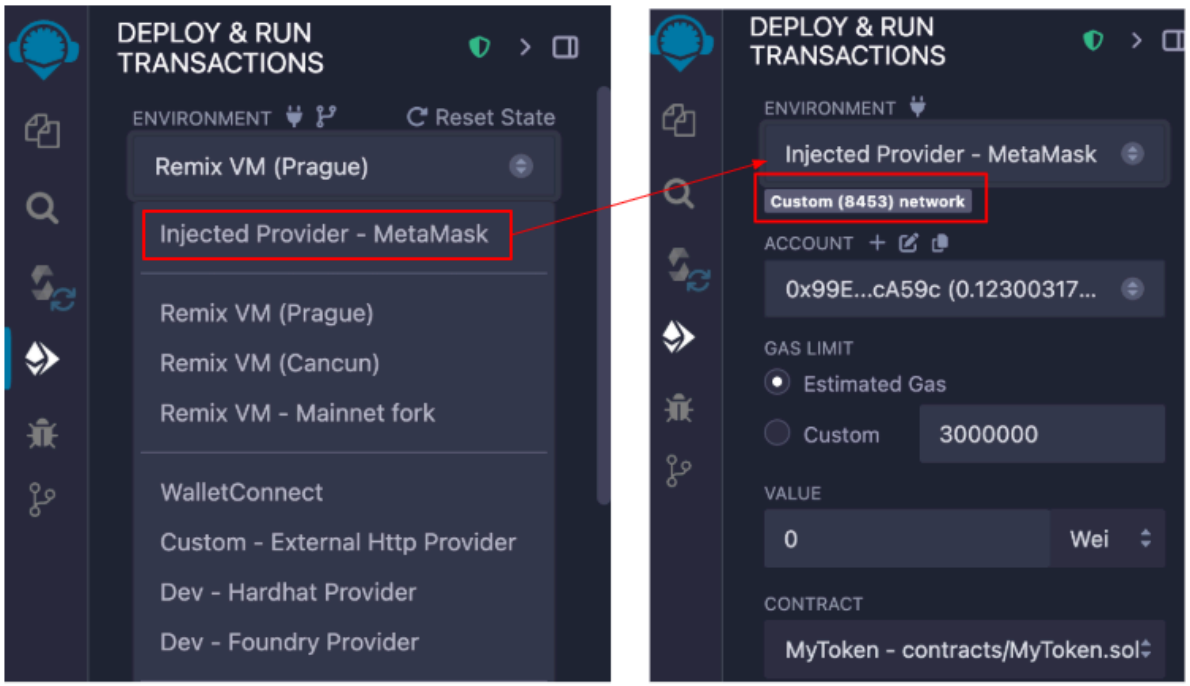
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

/// @title MyStablecoin - ERC-20 Stablecoin Example
contract MyStablecoin is ERC20, Ownable {
    constructor(string memory name, string memory symbol)
        ERC20(name, symbol)
        Ownable(msg.sender)
    {}

    /// @notice Mint tokens to an address (onlyOwner)
    function mint(address to, uint256 amount) external onlyOwner {
        require(to != address(0), "Cannot mint to zero address");
        _mint(to, amount);
    }

    /// @notice Burn tokens from an address (onlyOwner)
    function burn(address from, uint256 amount) external onlyOwner {
        require(from != address(0), "Cannot burn from zero address");
        _burn(from, amount);
    }
}
```

실제 세상에서 가장 유통량이 많은 스테이블코인인 USDT도 ERC-20 토큰입니다. 컨트랙트 코드를 보면 단순합니다. 그런 단순한 코드가 현재 $157B의 가치를 가진다는게 놀랍지 않나요? 하지만, USDT는 이더리움에서만 발행된게 아니고, 수십개의 블록체인 위에서 발행되어 있습니다. 그래서 각 체인에서의 구현 방식은 조금씩 다를 수 있습니다.

7. 내가 구현한 **ERC-20** 토큰을 **Base Sepolia** 테스트넷에 발행하고 사용해 보기
5번 섹션에서 구현한 토큰을 Remix 가상환경이 아니라 실제 블록체인에 발행할 수 있습니다. 이번에는 Base Sepolia 테스트넷 위에 당신이 만든 ERC-20 토큰을 발행해 봅시다. 우선 Metamask에 로그인해서 네트워크에서 Base Sepolia 테스트넷을 선택하고 컨트랙트를 배포할 주소를 선택하세요. 이 주소가 컨트랙트의 Owner가 될 것입니다. 그 다음 Remix Deploy & Run 탭에서 Environment를 Injected Provider - Metamask 로 선택하면 Metamask와 연결을 하고 Account에 당신의 주소가 보일 것입니다. 여기에 Metamask 주소가 보이지 않으면 지갑과 연결되지 않은 것입니다. 그리고 Chain ID가 원하는 체인의 ID로 나왔는지 확인하세요. 이제 Deploy에서 입력 변수들을 설정하고 transact 버튼을 눌러 배포하면 됩니다.

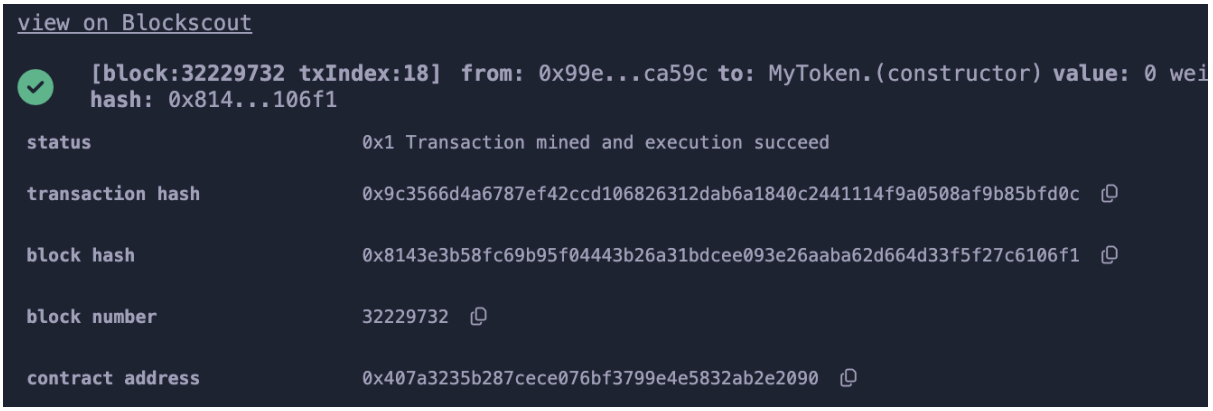


Base Sepolia Testnet으로 연결한 경우 (Chain ID = 84532)



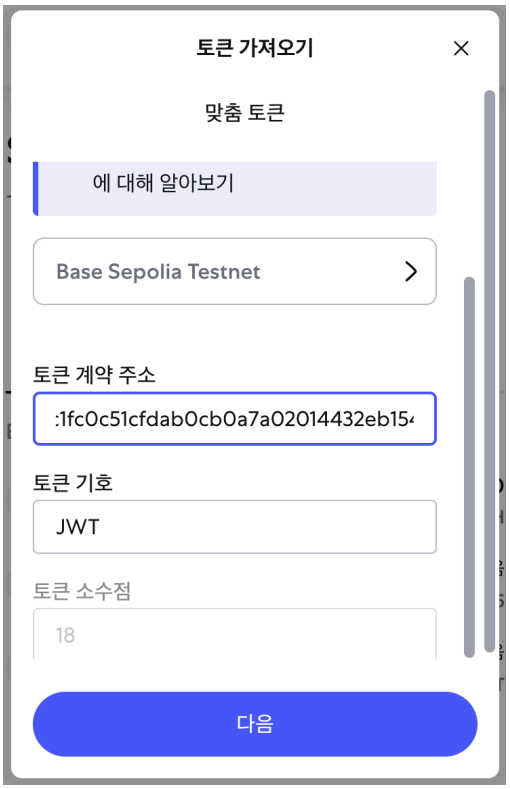
Base Mainnet으로 연결한 경우 (Chain ID = 8453)

이제 실제 블록체인 환경이기 때문에 가스비를 지불해야 합니다. Remix에서 가스비 경고가 뜨더라도 일단 실행해 보세요. 그러면 Metamask에서 적절한 가스비로 설정해 체인에 트랜잭션을 보낼 것입니다. Metamask에서 최종 Confirm을 하면 컨트랙트 생성 트랜잭션이 베이스 체인에 보내진 것을 Remix console 창에서 확인할 수 있습니다. 아래 그림과 같이 로그를 펼쳐보면 contract address를 복사할 수 있습니다. 이 주소를 Sepolia Basescan에 조회해 보면 컨트랙트가 잘 배포된 것을 확인할 수 있습니다.



자, 이제 당신은 실제 블록체인에 배포된 진짜 토큰을 가지게 되었습니다. mint() 함수를 호출해서 당신에게 토큰을 민팅해 보세요. Remix의 장점은 별도의 프론트앤드 없이 Remix에서 모든 함수를 호출할 수 있다는 것입니다. mint() 함수가 잘 실행되었는지 Basescan을 통해 확인해 보세요. 그리고 당신의 지갑에

토큰이 잘 들어왔는지 확인해 보세요. 이를 위해선 먼저 당신이 만든 토큰을 Metamask에 인식시켜야 하겠죠.



이 후 지갑에 토큰이 잘 들어왔는지 확인해 보세요. 그런데, 개수가 생각한 것보다 너무 적나요? 여기서 Decimal 개념이 등장합니다. Solidity에서는 부동소수점(float, double)이 없기 때문에 모든 수치는 정수 단위(uint)로 저장되고 계산됩니다. 그래서 소수 단위를 정수로 표현하기 위해 decimals를 도입했습니다. decimals = 18 이면 아래와 같이 토큰의 개수를 표현하는 것이죠.

- 1.0 토큰 = 1 * 10^18 = 1000000000000000000 (정수)
- 0.5 토큰 = 0.5 * 10^18 = 500000000000000000

따라서, 1개 토큰을 보낼 때 입력값으로 1000000000000000000를 넣어줘야 합니다. Remix 내에서만 테스트할 때는 이슈가 없었겠지만(실제로는 소수점 단위의 토큰을 주고 받음) Metamask에서는 decimals를 계산해서 표시하는 것입니다. 이더리움이 decimals로 18을 사용하기 때문에 대부분의 토큰이 18로 지정합니다. 물론 다른 값으로 지정해도 됩니다. 매번 입력할 때마다 계산하기 복잡하기 때문에 Unit Converter 페이지를 띄워두고 입력하려는 값(Ether)를 decimals를 적용한 값(Wei)로 바꿔서 복사해서 쓰는게 에러를 줄이는 방법입니다.

- 10 Ether = 10000000000000000000 Wei

당신의 토큰이 체인에 정상적으로 배포되었고, 당신의 주소로 토큰을 잘 받았다면, 이제 이 토큰을 다른 사람에게 전송해 보세요. 지갑에서 실행해도

되고, Remix에서 transfer 함수로 실행해도 됩니다. 물론 다른 사람의 지갑에서 당신의 토큰을 인식시키기 위해 컨트랙트 주소를 알려주어야 합니다.