EXPERIMENT 1

Write a program to generate public and private key using OpenSSL.

```
import subprocess


def generate_rsa_keys(private_key_file="private.pem", public_key_file="public.pem", bits=2048):
    # Generate private key
    subprocess.run([
        "openssl", "genpkey",
        "-algorithm", "RSA",
        "-pkeyopt", f"rsa_keygen_bits:{bits}",
        "-out", private_key_file
    ], check=True)

    # Generate public key from private key
    subprocess.run([
        "openssl", "pkey",
        "-in", private_key_file,
        "-pubout",
        "-out", public_key_file
    ], check=True)

    # Read and print the public key
    with open(public_key_file, "r") as f:
        public_key_content = f.read()

    print(f"Keys generated successfully ")
    print(f"Private key saved in: {private_key_file}")
    print(f"Public key saved in: {public_key_file}\n")
```

```
    print(" Public Key (PEM format):\n")
    print(public_key_content)


# Example usage
generate_rsa_keys()
```

OUTPUT:

```
Keys generated successfully ✅
Private key saved in: private.pem
Public key saved in: public.pem

🔑 Public Key (PEM format):

-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwvaG43/4RJoQOxApDqvU
NBd2KBi3E6GwsZJ6Y8KdneTOqC8q/BRyRZEscm1uay569dkHEKB5gYpUjTU9Jaqm
nwmo8cIJPtMH9Ic8A8c2rY+8EQYwihf/EGIQhLypHzudKPjvskUSgsU5a8/lKKB4
JcaUwUhQ9JUTZRbLz+GxdW54ZZFcz1Ygt4TzrcA6VpbabGEX5U/lG+bD3U5g8IDL
CAz0FADpoiJZcFpX5MK9IbBhH1GOfp4tZ8aWkfqO96OnIn4pOjOlZMnwpjreHF6n
oywlDkpXfooMKoMEwVfNyi6tK9IwR7qgItFmlSGou0CCrFhjoTQIpUuu9KQQ2/2g
AwIDAQAB
-----END PUBLIC KEY-----
```

EXPERIMENT 2

Write a program to create a simple Blockchain using Python.

```
import hashlib
import time


class Block:
    def __init__(self, index, timestamp, data, previous_hash):
        self.index = index
        self.timestamp = timestamp
        self.data = data
        self.previous_hash = previous_hash
        self.hash = self.calculate_hash()

    def calculate_hash(self):
        """Generate SHA-256 hash for the block."""
        block_string = str(self.index) + str(self.timestamp) + str(self.data) + str(self.previous_hash)
        return hashlib.sha256(block_string.encode()).hexdigest()



class Blockchain:
    def __init__(self):
        # create the first block (genesis block)
        self.chain = [self.create_genesis_block()]

    def create_genesis_block(self):
        """Manually create the first block (Genesis Block)."""
        return Block(0, time.ctime(), "Genesis Block", "0")
```

```
    def get_latest_block(self):
        return self.chain[-1]

    def add_block(self, new_data):
        """Add a block with new data to the chain."""
        prev_block = self.get_latest_block()
        new_block = Block(len(self.chain), time.ctime(), new_data, prev_block.hash)
        self.chain.append(new_block)

    def is_chain_valid(self):
        """Check the validity of the blockchain."""
        for i in range(1, len(self.chain)):
            current = self.chain[i]
            prev = self.chain[i-1]

            if current.hash != current.calculate_hash():
                print("Block", i, "has been tampered!")
                return False
            if current.previous_hash != prev.hash:
                print("Block", i, "previous hash does not match!")
                return False
        return True


# ---------- Example usage ----------
my_chain = Blockchain()
my_chain.add_block("Alice pays Bob 10 coins")
my_chain.add_block("Bob pays Charlie 5 coins")
```

```
my_chain.add_block("Charlie pays Dave 2 coins")


# Print blockchain
for block in my_chain.chain:
    print("Index:", block.index)
    print("Timestamp:", block.timestamp)
    print("Data:", block.data)
    print("Hash:", block.hash)
    print("Previous Hash:", block.previous_hash)
    print("-" * 50)


# Validate blockchain
print("Is blockchain valid?", my_chain.is_chain_valid())
```

OUTPUT:

```
Index: 0
Timestamp: Tue Sep  2 22:17:10 2025
Data: Genesis Block
Hash: ce0ca05cec2eb9dc4409d26e76187dc16cc2368f1ab420896b5830a01ef1569d
Previous Hash: 0
--------------------------------------------------
Index: 1
Timestamp: Tue Sep  2 22:17:10 2025
Data: Alice pays Bob 10 coins
Hash: fec2768a0b5168be0d3bebb9d11e236f43bf65e9333f6baceb5f4bac219d8886
Previous Hash: ce0ca05cec2eb9dc4409d26e76187dc16cc2368f1ab420896b5830a01ef1569d
--------------------------------------------------
Index: 2
Timestamp: Tue Sep  2 22:17:10 2025
Data: Bob pays Charlie 5 coins
Hash: d4a5709313e86a0e14ea7c898750e0ca763a5c4d4703ed8e3cbd66fe08585330
Previous Hash: fec2768a0b5168be0d3bebb9d11e236f43bf65e9333f6baceb5f4bac219d8886
--------------------------------------------------
Index: 3
Timestamp: Tue Sep  2 22:17:10 2025
Data: Charlie pays Dave 2 coins
Hash: 567a7a83753e4945c97501f77b2553d1c1f86633628fc24ac53d241bfdbc4616
Previous Hash: d4a5709313e86a0e14ea7c898750e0ca763a5c4d4703ed8e3cbd66fe08585330
--------------------------------------------------
Is blockchain valid? True
```

Alternate code to include Merkle root and nonce

```
import hashlib

import time

import math


def calculate_merkle_root(transactions):

    """Calculate Merkle Root from a list of transactions."""

    if not transactions:

        return hashlib.sha256("".encode()).hexdigest()


    # Start with transaction hashes

    tx_hashes = [hashlib.sha256(tx.encode()).hexdigest() for tx in transactions]


    while len(tx_hashes) > 1:

        if len(tx_hashes) % 2 != 0:  # if odd, duplicate last element

            tx_hashes.append(tx_hashes[-1])


        new_level = []

        for i in range(0, len(tx_hashes), 2):

            combined = tx_hashes[i] + tx_hashes[i+1]

            new_level.append(hashlib.sha256(combined.encode()).hexdigest())

        tx_hashes = new_level


    return tx_hashes[0]



class Block:
```

```
def __init__(self, index, timestamp, transactions, previous_hash, difficulty=2):
    self.index = index
    self.timestamp = timestamp
    self.transactions = transactions  # list of transactions
    self.previous_hash = previous_hash
    self.nonce = 0
    self.merkle_root = calculate_merkle_root(transactions)
    self.hash = self.mine_block(difficulty)

def calculate_hash(self):
    block_string = (
        str(self.index) +
        str(self.timestamp) +
        str(self.transactions) +
        str(self.previous_hash) +
        str(self.nonce) +
        str(self.merkle_root)
    )
    return hashlib.sha256(block_string.encode()).hexdigest()

def mine_block(self, difficulty):
    """Proof-of-Work: adjust nonce until hash starts with leading zeros."""
    prefix = "0" * difficulty
    while True:
        hash_value = self.calculate_hash()
        if hash_value.startswith(prefix):
            return hash_value
        else:
```

```
            self.nonce += 1



class Blockchain:
    def __init__(self, difficulty=2):
        self.chain = [self.create_genesis_block()]
        self.difficulty = difficulty

    def create_genesis_block(self):
        return Block(0, time.ctime(), ["Genesis Block"], "0")

    def get_latest_block(self):
        return self.chain[-1]

    def add_block(self, transactions):
        prev_block = self.get_latest_block()
        new_block = Block(len(self.chain), time.ctime(), transactions, prev_block.hash, self.difficulty)
        self.chain.append(new_block)

    def is_chain_valid(self):
        for i in range(1, len(self.chain)):
            current = self.chain[i]
            prev = self.chain[i-1]

            if current.hash != current.calculate_hash():
                print("Block", i, "hash mismatch!")
                return False
            if current.previous_hash != prev.hash:
```

```
            print("Block", i, "previous hash mismatch!")
            return False
        return True



# ---------- Example usage ----------
my_chain = Blockchain(difficulty=3)
my_chain.add_block(["Alice pays Bob 10 coins", "Bob pays Charlie 5 coins"])
my_chain.add_block(["Charlie pays Dave 2 coins", "Dave pays Eve 1 coin"])

# Print blockchain
for block in my_chain.chain:
    print("\nIndex:", block.index)
    print("Timestamp:", block.timestamp)
    print("Transactions:", block.transactions)
    print("Merkle Root:", block.merkle_root)
    print("Nonce:", block.nonce)
    print("Hash:", block.hash)
    print("Previous Hash:", block.previous_hash)
    print("-" * 60)

# Validate chain
print("\nIs blockchain valid?", my_chain.is_chain_valid())
```

OUTPUT:

```
Index: 0
Timestamp: Tue Sep  2 23:18:16 2025
Transactions: ['Genesis Block']
Merkle Root: 89eb0ac031a63d2421cd05a2fbe41f3ea35f5c3712ca839cbf6b85c4ee07b7a3
Nonce: 172
Hash: 002df3d1a599c74f5261f68d84420741e7a76ac3cff3d036b9b22ec8c6129f16
Previous Hash: 0
------------------------------------------------------------

Index: 1
Timestamp: Tue Sep  2 23:18:16 2025
Transactions: ['Alice pays Bob 10 coins', 'Bob pays Charlie 5 coins']
Merkle Root: 8e2a53202f0c8415e7c25cdc6c82f0b3085401b6ec713149fdfc1921071238e5
Nonce: 1234
Hash: 00084d813bbb5e4e07c7770f342e4d1f9910785b8adf46a729c7a6decac3029b
Previous Hash: 002df3d1a599c74f5261f68d84420741e7a76ac3cff3d036b9b22ec8c6129f16
------------------------------------------------------------

Index: 2
Timestamp: Tue Sep  2 23:18:17 2025
Transactions: ['Charlie pays Dave 2 coins', 'Dave pays Eve 1 coin']
Merkle Root: d87426eb9e3157f2b13daad77782b1476822f469829fa00330491347338adaf8
Nonce: 1424
Hash: 0006d9738a1a1539ecfd8adeff13e546b10f413d780cd7ed91fcba46c86b5fb2
Previous Hash: 00084d813bbb5e4e07c7770f342e4d1f9910785b8adf46a729c7a6decac3029b
------------------------------------------------------------

Is blockchain valid? True
```

EXPERIMENT 3

Develop and test smart contract on local Blockchain.

Deploying a Smart Contract to a Local Ganache Blockchain

Objective

To successfully deploy Solidity smart contract from the Remix web IDE to a local Ganache blockchain instance. This lab covers the complete workflow, including critical configuration of the EVM version to resolve common deployment failures and interpretation of post-deployment log messages.

Materials Required

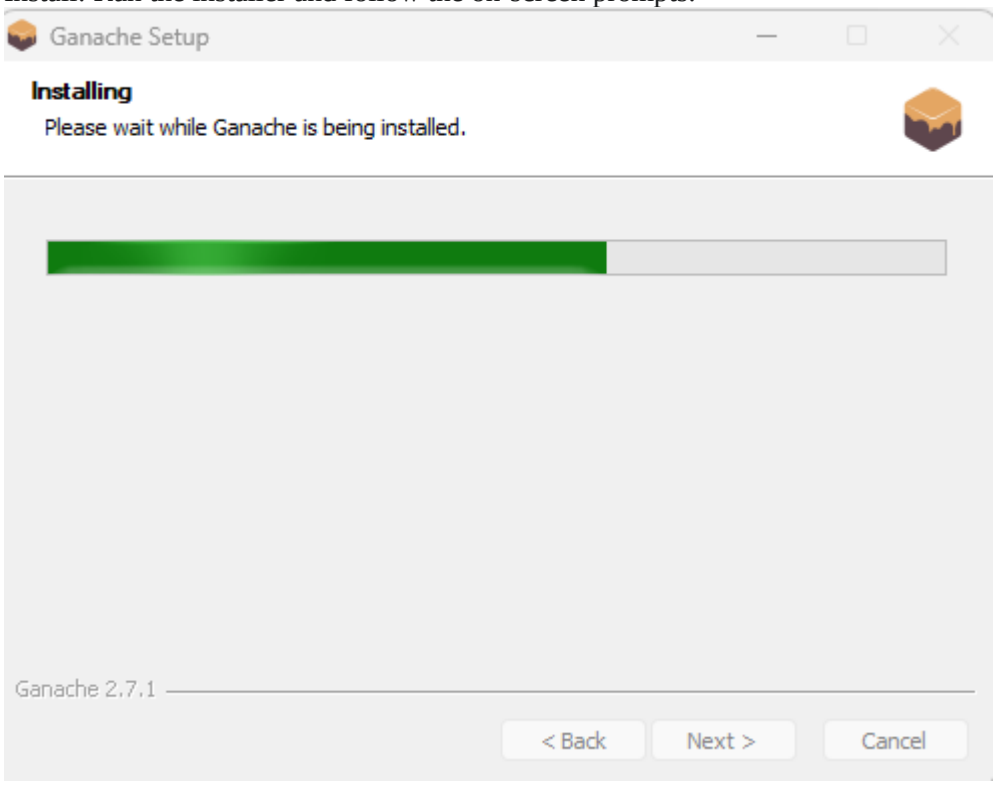
- A modern web browser (e.g., Google Chrome, Firefox)
- Internet access (to use Remix IDE)
- Ganache UI (Desktop Application)

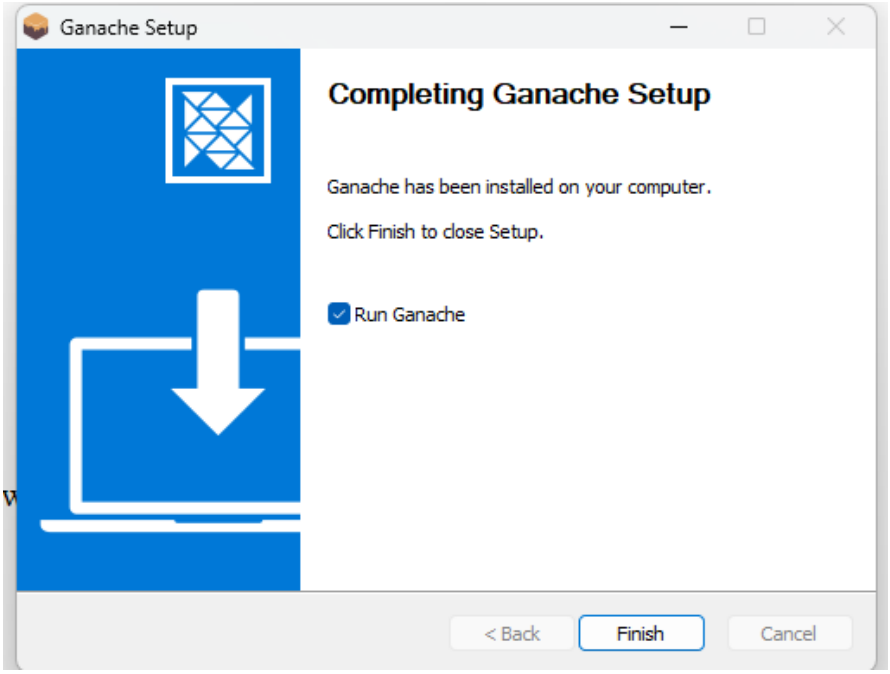
Procedure

Part 1: Install and Launch Ganache (The Local Blockchain)

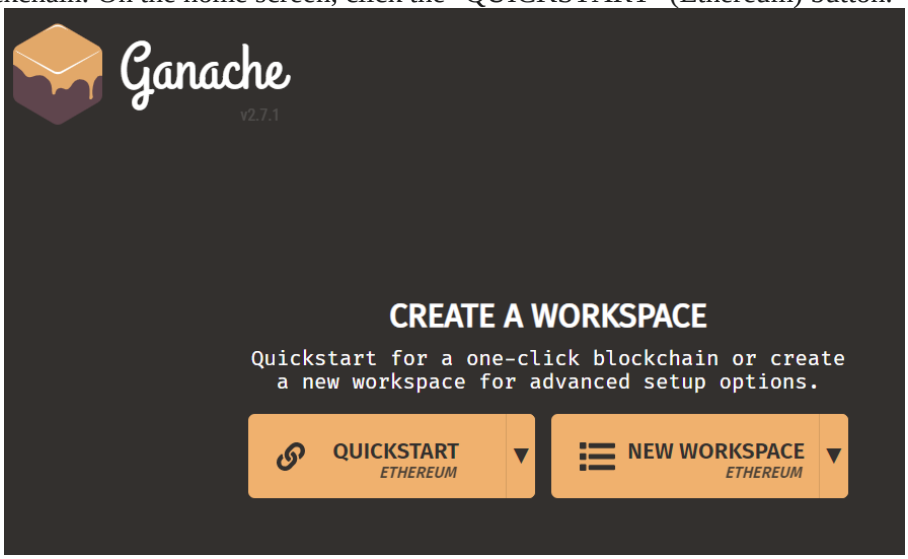
Download: Navigate to the https://archive.trufflesuite.com/ganache/ or https://drive.google.com/drive/folders/1fWhY-ELA9Hq1br_gezVSYBLXtwsw-4Ed?usp=sharing or https://github.com/ConsenSys-archive/ganache-ui/releases
and download the Ganache UI (desktop application) for your operating system.

1. Install: Run the installer and follow the on-screen prompts.

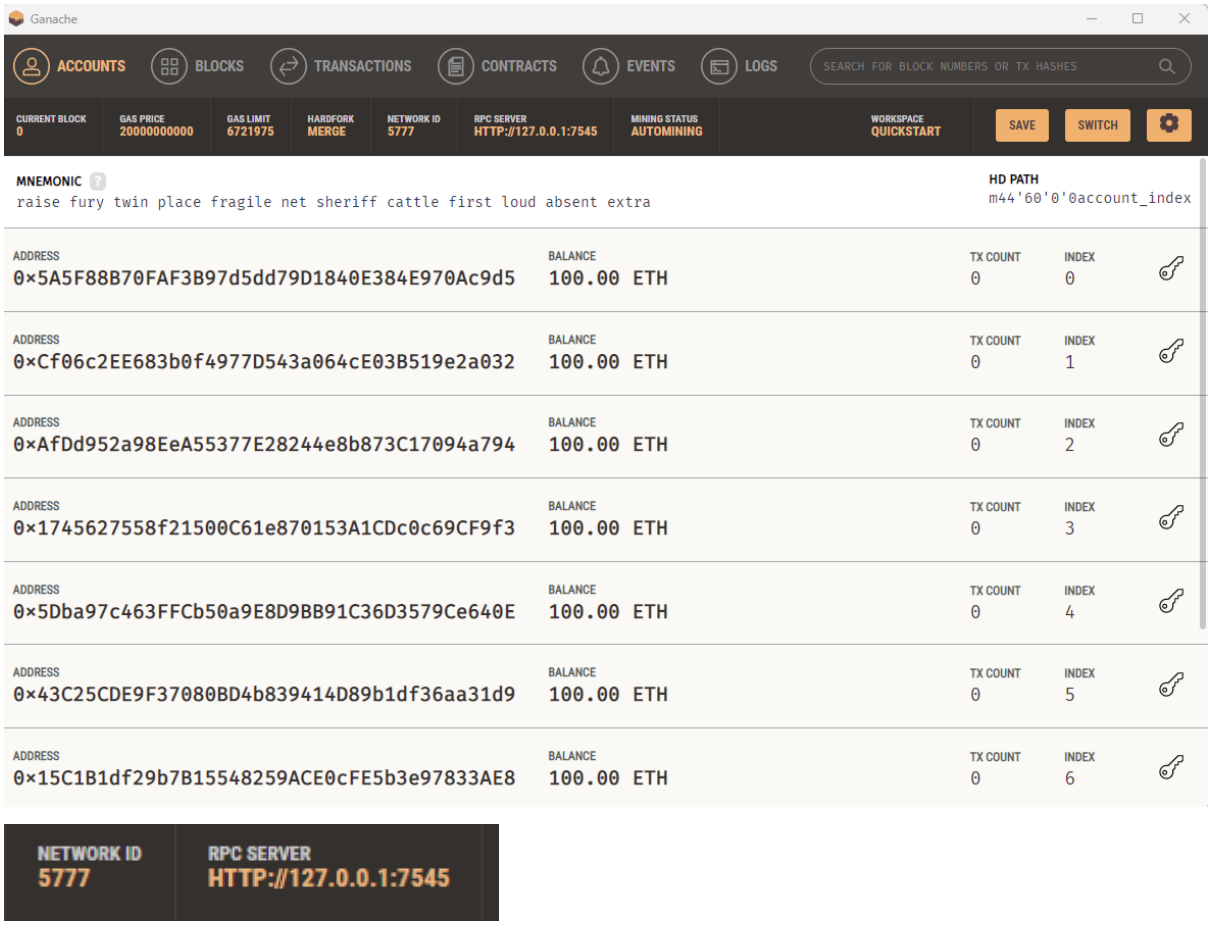




2. Launch Ganache: Open the Ganache application.
3. Start Blockchain: On the home screen, click the "QUICKSTART" (Ethereum) button.



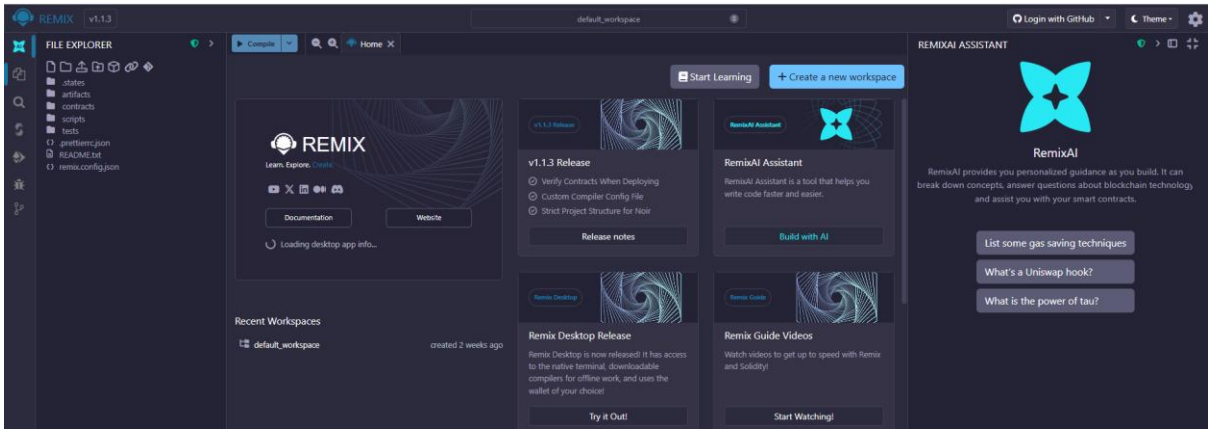
4. Observe: Your local blockchain is now running. Take note of two crucial pieces of information from the main screen:
   - RPC SERVER: http://127.0.0.1:7545 (This is the "address" of your local blockchain).
   - NETWORK ID: 5777 (This is the unique identifier for your private chain).



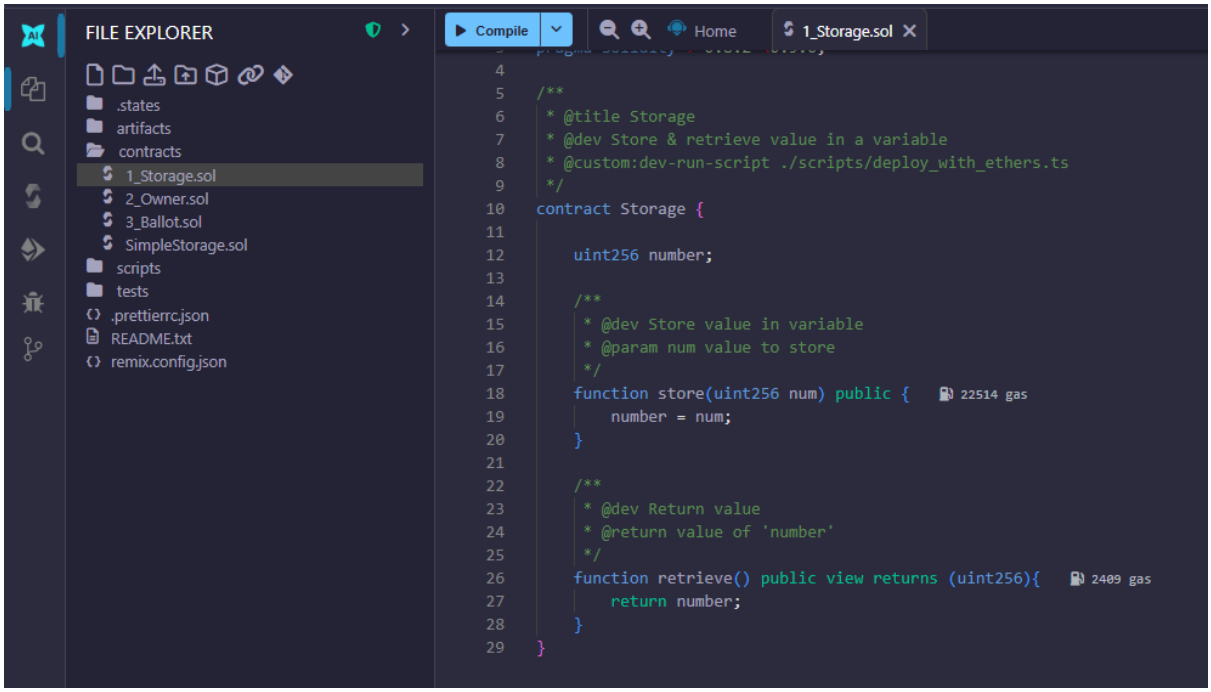
Leave Ganache running in the background.

Part 2: Prepare and Compile the Contract in Remix (The IDE)

1. **Open Remix:** In your web browser, navigate to remix.ethereum.org.



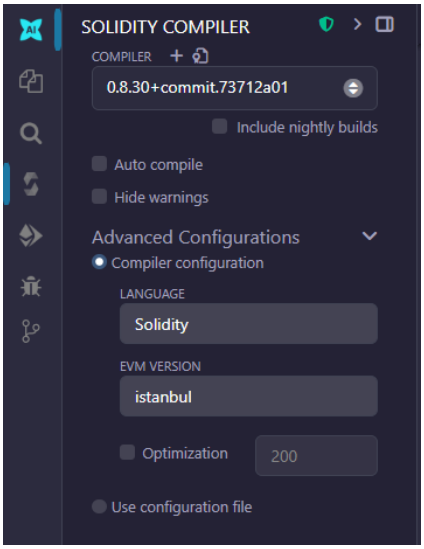
2. **Create File:** In the "File Explorer" tab on the left, create a new file named Storage.sol.

3. 

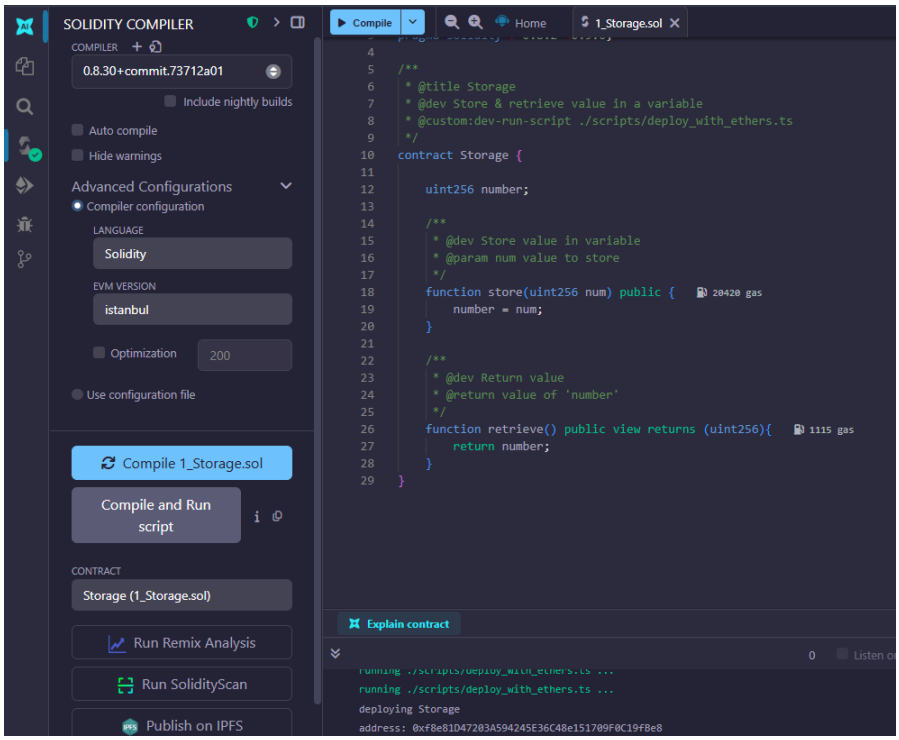
4. **Add Code:** Paste the following simple storage contract into the new file:

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

contract Storage {
    uint256 public myNumber;

    function store(uint256 _num) public {
        myNumber = _num;
    }

    function retrieve() public view returns (uint26) {
        return myNumber;
    }
}
```

5. **Navigate to Compiler:** Select the "Solidity compiler" tab (the second icon on the far-left sidebar).
6. **Select Compiler Version:** In the "COMPILER" dropdown, select a version that matches the pragma (e.g., 0.8.20 or any 0.8.x version).
7. **CRITICAL - Set EVM Version:**
   - Click on the **"Advanced Configurations"** link (under the "Compile" button).
   - Find the **"EVM VERSION"** dropdown.
   - Change this value from compiler default to **istanbul**.

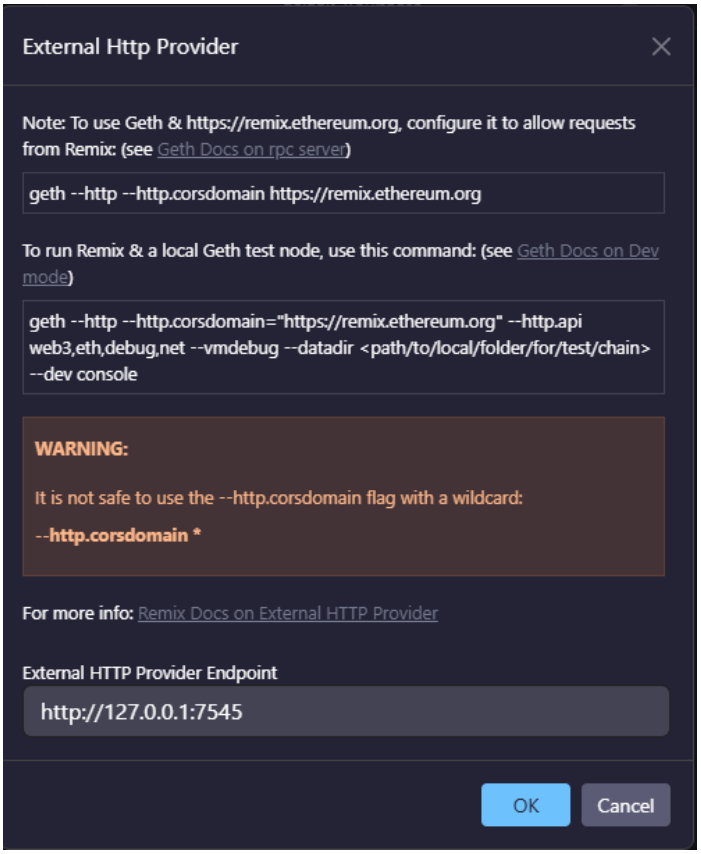


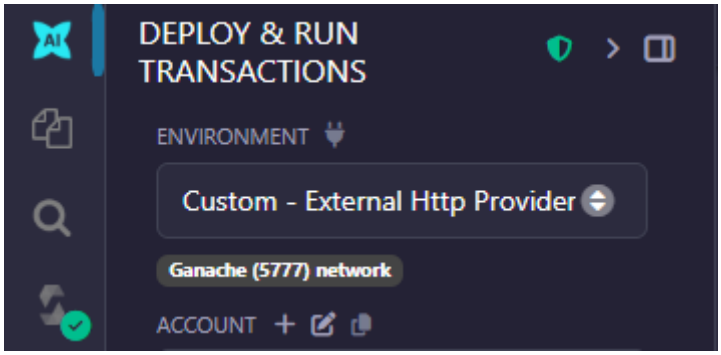
- Click the blue **"Compile Storage.sol"** button. A green checkmark will appear on the compiler icon if successful.



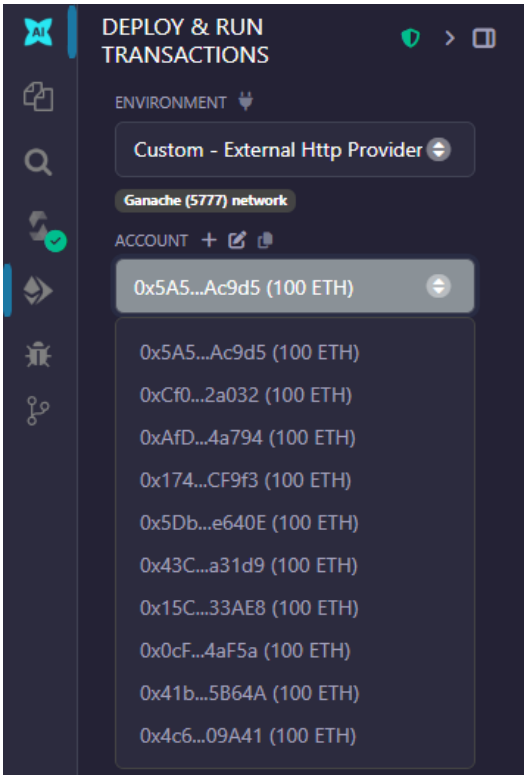
Part 3: Connect Remix to Ganache (The "External HTTP Provider")

1. **Navigate to Deploy Tab:** Select the "Deploy & run transactions" tab (the third icon on the far-left sidebar).
2. **Select Environment:** At the top of the deploy panel, click the "ENVIRONMENT" dropdown. By default, it says "Remix VM (Shanghai)".
3. **Choose Provider:** Select **"External HTTP Provider"**.
4. Enter Endpoint: A pop-up box will appear asking for the "External HTTP Provider Endpoint". Enter the RPC Server URL you noted from Ganache: http://127.0.0.1:7545



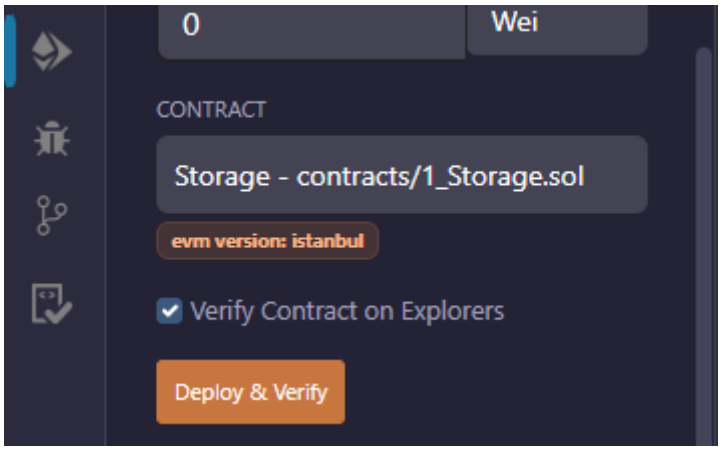


5. **Confirm Connection:** Click "OK". You should now see the "ACCOUNT" dropdown below populate with the 10 accounts from your Ganache application, each showing a balance of 100 ETH.

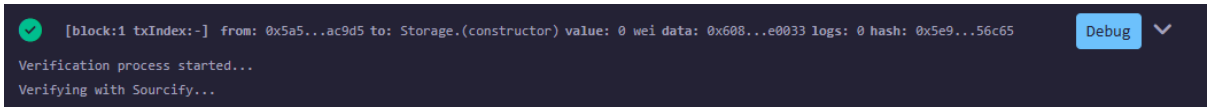


Part 4: Deploy and Verify the Contract

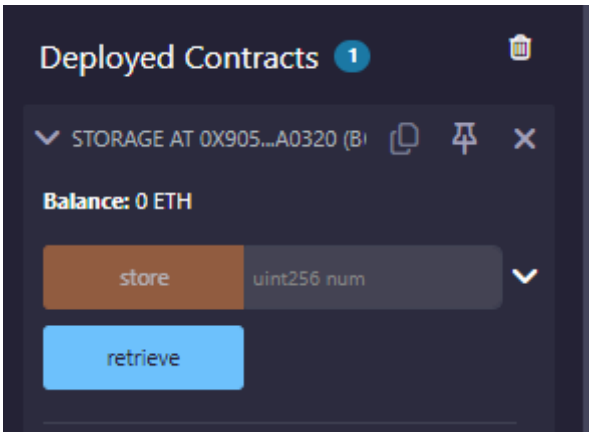
1. **Deploy:** In the "Deploy & run transactions" tab, ensure your Storage contract is selected in the "CONTRACT" dropdown. Click the orange **"Deploy"** button.

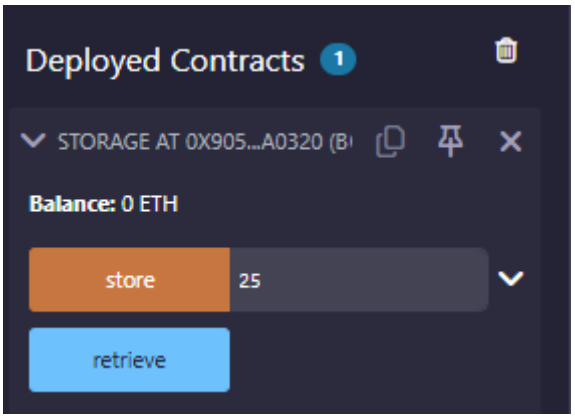


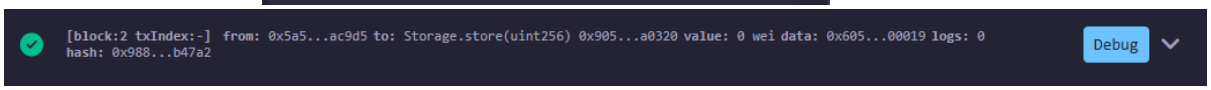
2. **Observe Remix Terminal:** The terminal at the bottom of the Remix window will show the transaction details. You should see a green checkmark and a log similar to this, confirming success:
   [block:1 txIndex:-]
   from: 0x...
   to: Storage.(constructor)
   ...
   hash: 0x...
   status: 0x1 Transaction mined and execution succeed

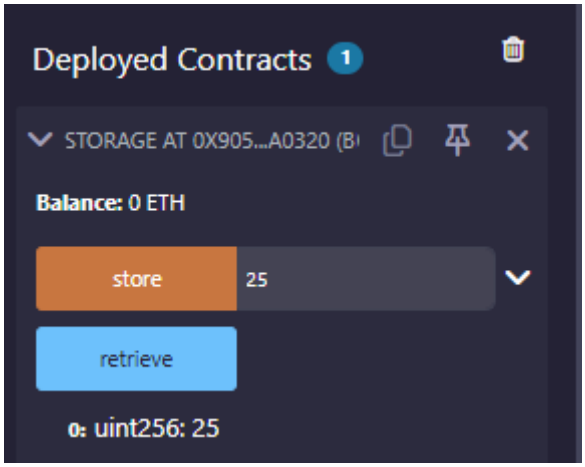


3. **Verify Deployment:** In the "Deployed Contracts" section at the bottom of the Remix deploy panel, you will see your Storage contract.
    - Click the retrieve button: it will return 0.
    - Enter a number (e.g., 42) into the store function's input field and click the store button.
    - Click retrieve again. It will now return 42.

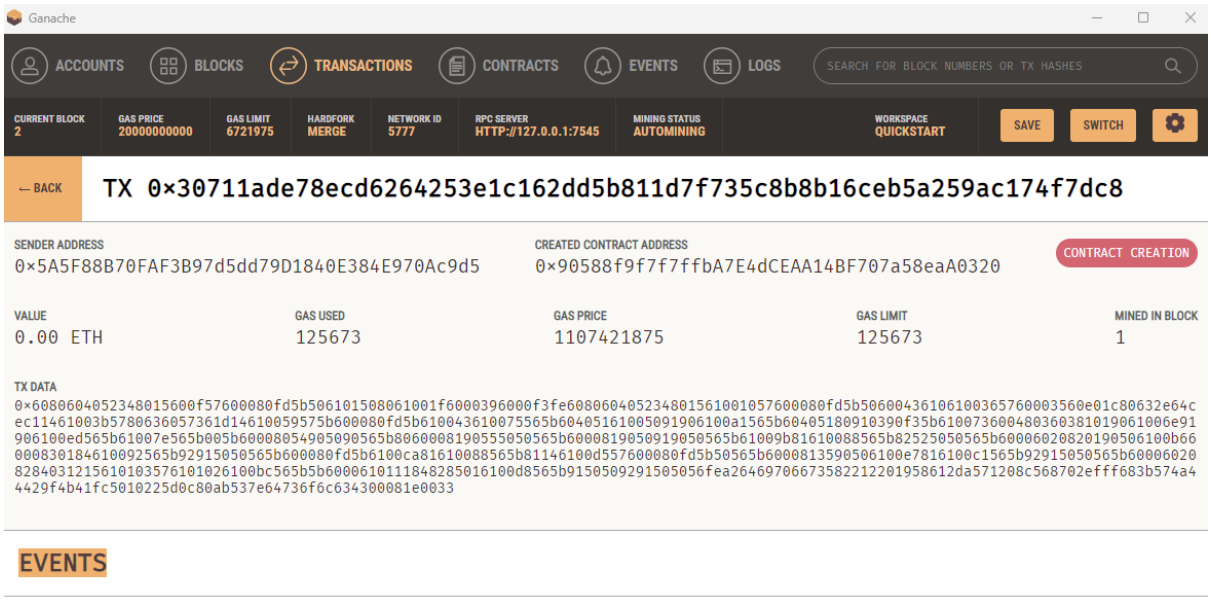








4. **Verify in Ganache:** Look at your Ganache application. The "TRANSACTIONS" tab will now list two transactions: one "Contract Creation" and one "Contract Call".



4. Key Concepts & Analysis (Troubleshooting Explained)

This section explains *why* we performed certain steps and what the log messages mean.

Analysis 1: The "Gas estimation errored..." Error & EVM Mismatch

- **The Problem:** Before this lab, you saw a "Gas estimation errored" message. This error means the contract's constructor failed during simulation. Ganache couldn't even "guess" the gas cost because the contract was fundamentally invalid on its network.
- **The Cause: EVM Version Mismatch**
  - **EVM (Ethereum Virtual Machine):** This is the "operating system" that runs smart contract code. The EVM is updated over time with new features and instructions (opcodes) via hard forks (e.g., istanbul, paris, shanghai).
  - **The Compiler (Remix):** Modern Solidity compilers (like v0.8.x) generate bytecode using the *newest* opcodes for the *newest* EVMs (e.g., shanghai).
  - **The Blockchain (Ganache):** Your Ganache UI version runs an *older* EVM version, in your case, **istanbul**.
  - **The Mismatch:** The istanbul EVM does not understand the modern opcodes (like PUSH0) that the shanghai EVM uses. When Ganache tried to read the new bytecode from Remix, it hit an instruction it didn't recognize and immediately failed.
- The Solution (Part 2, Step 6):
  By manually setting the "EVM VERSION" in Remix to istanbul, you instructed the compiler to only generate older, compatible bytecode that the Ganache istanbul EVM can understand. This resolved the incompatibility.

Analysis 2: The "Sourcify Verification Failed" Message

- The Message: After your successful deployment, you saw:
  Verification process started... Verifying with Sourcify... Sourcify verification failed: Chain 5777 not found
- **This is not an error.** Your deployment was 100% successful.
- **What it means:**
  - **Sourcify:** This is a *public* database that tries to verify the source code of contracts on *public*

blockchains (like Ethereum Mainnet, Sepolia, etc.) to build trust.

- **The "Failure":** Sourcify's public servers tried to find a *public* chain with the Network ID **5777** (your Ganache chain). Since your chain is private and local (it only exists on your computer), Sourcify correctly reported "Chain 5777 not found."

Conclusion: This message is expected and simply confirms that you are working on a private, local test network that is not visible to the public internet.

EXPERIMENT 4

Develop and test smart contract on Ethereum test networks.

1. Objective

This lab will guide you through the complete lifecycle of a basic smart contract. You will learn to use the Remix IDE (a browser-based development environment) and the MetaMask wallet to write, compile, deploy, and interact with a smart contract on the Sepolia public testnet.

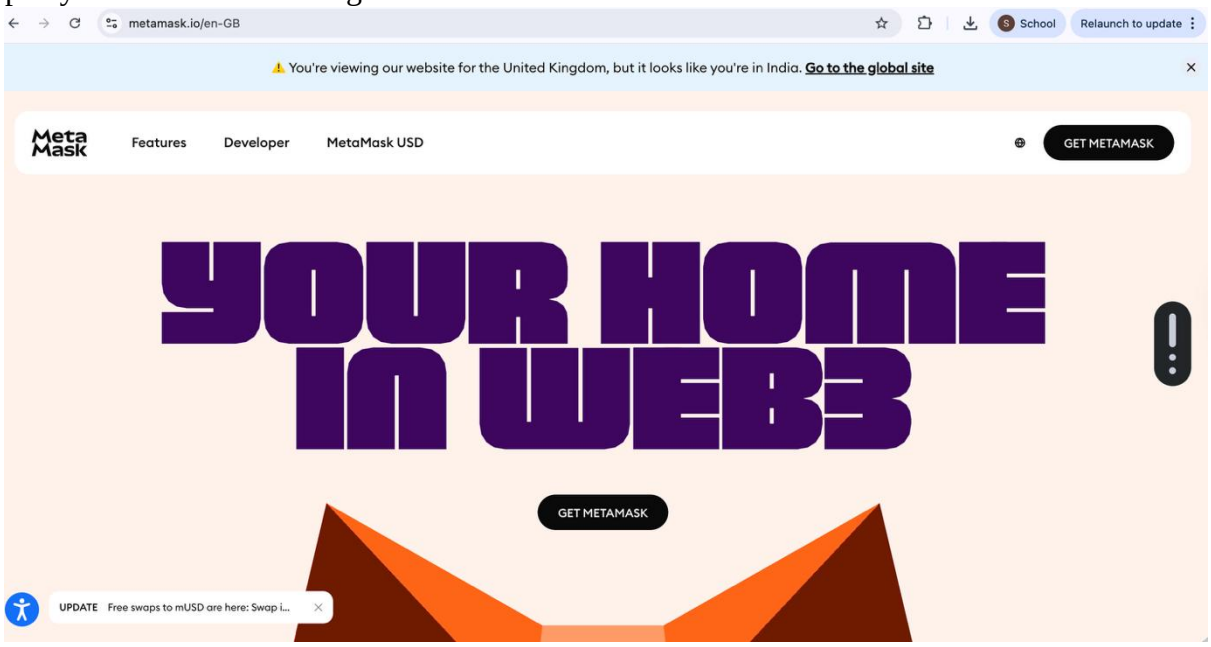
2. Prerequisites

- A computer with an internet connection.
- A modern web browser (e.g., Google Chrome, Firefox, Brave).

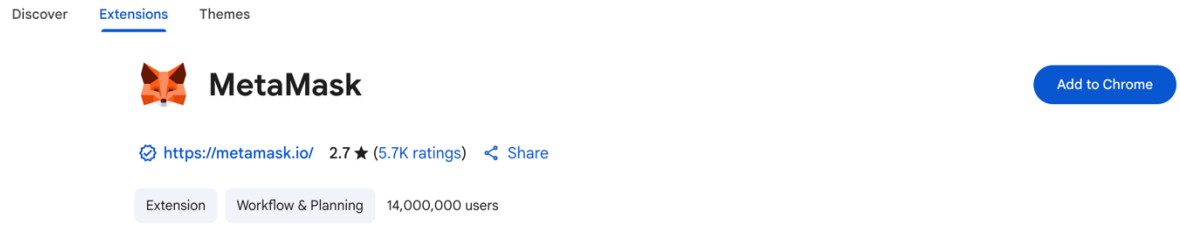
3. Procedure

Part A: Wallet Setup

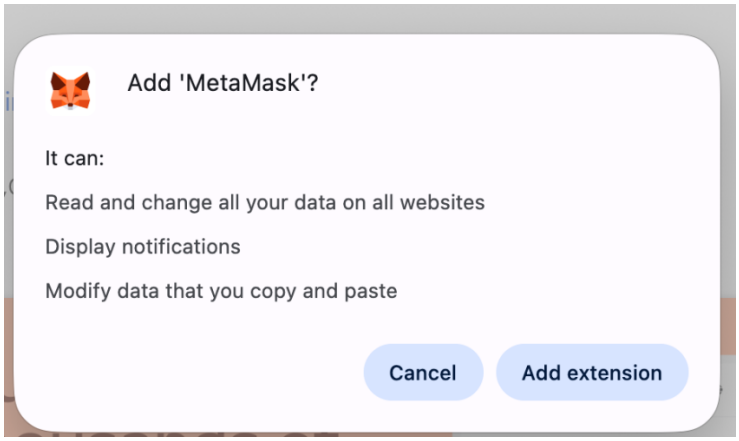
1. Install MetaMask:
   - Open your browser and navigate to the official website: metamask.io.

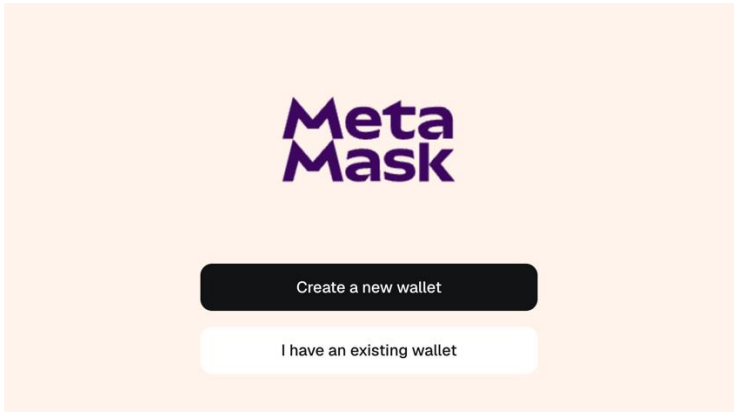


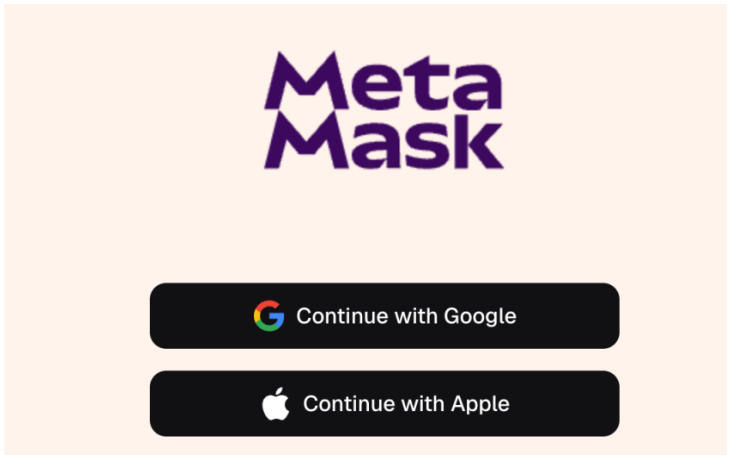
   - Download and install the MetaMask browser extension.



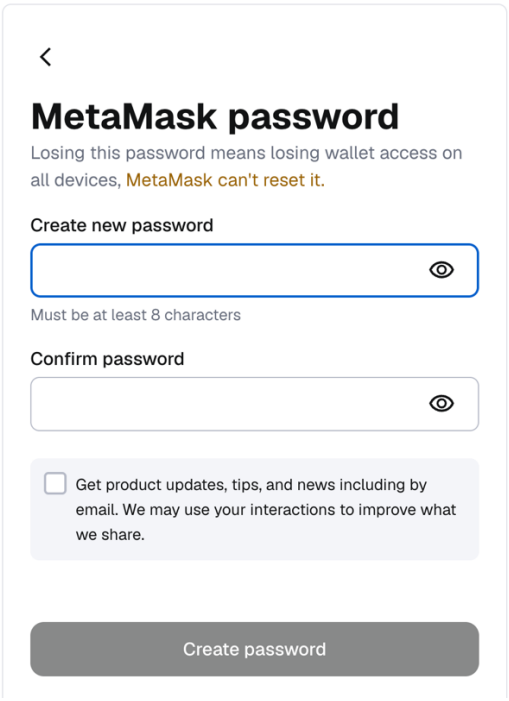
   - Follow the on-screen prompts to "Create a new wallet."



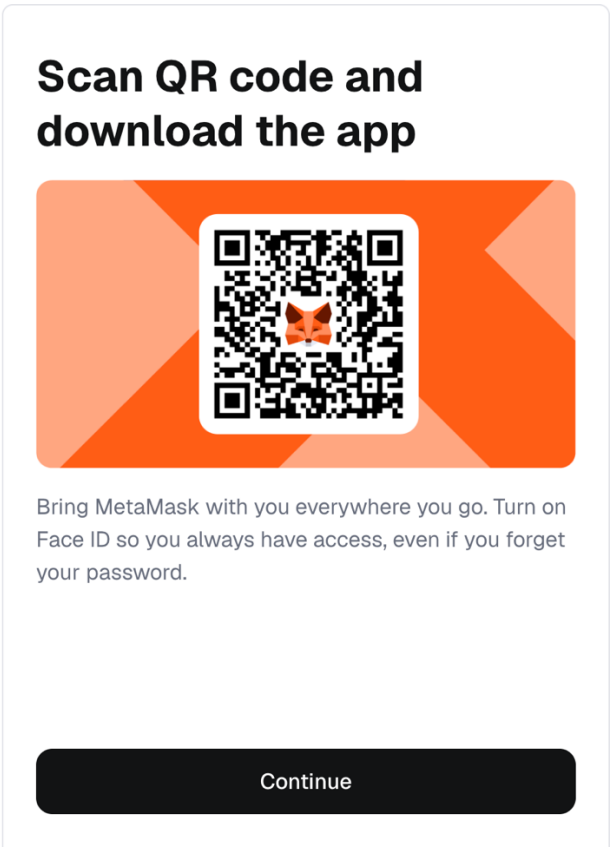


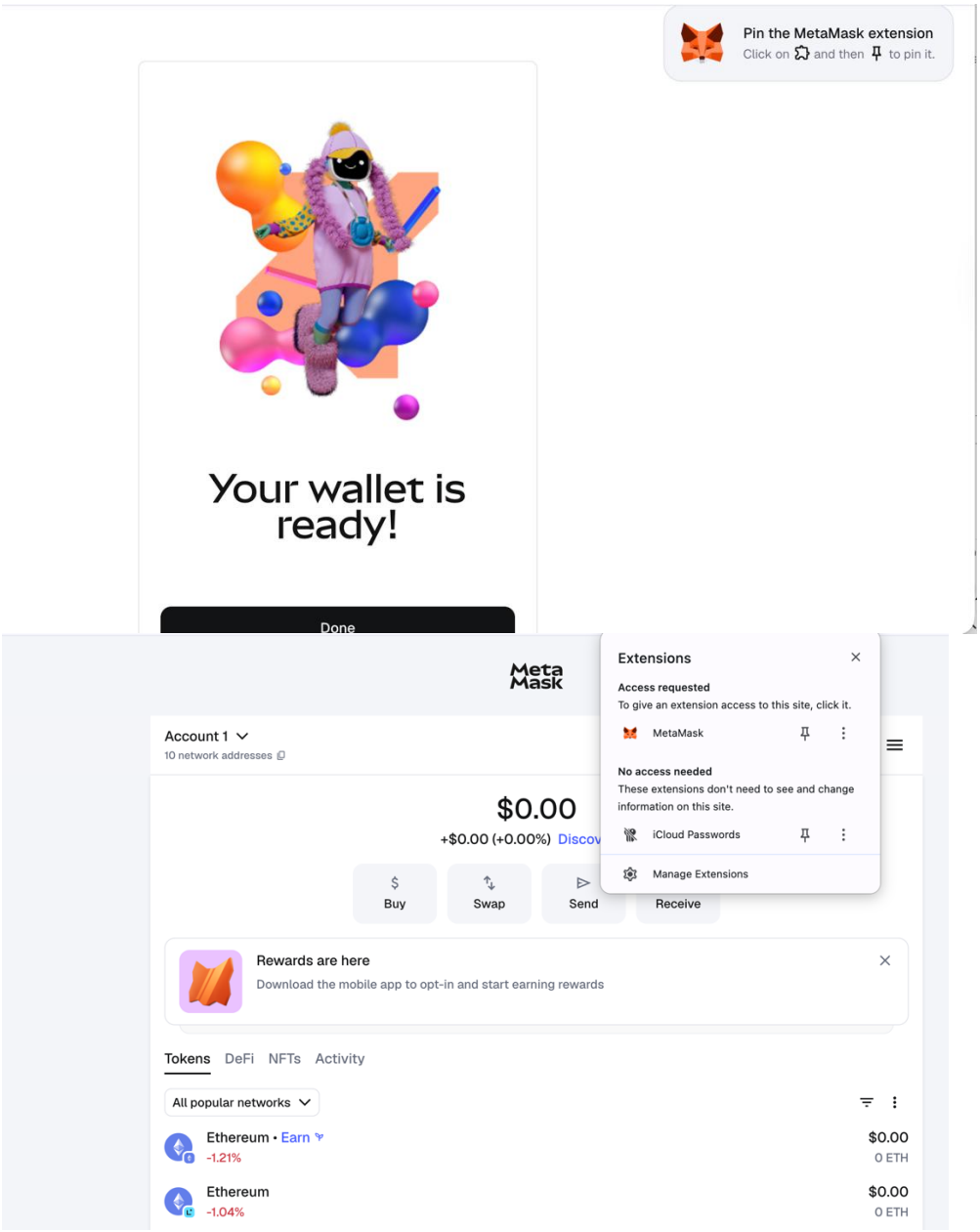


Sign in with your Gmail account and create a new password



You can scan the QR Code and have a Metamask app installed in your mobile phones otherwise click continue.



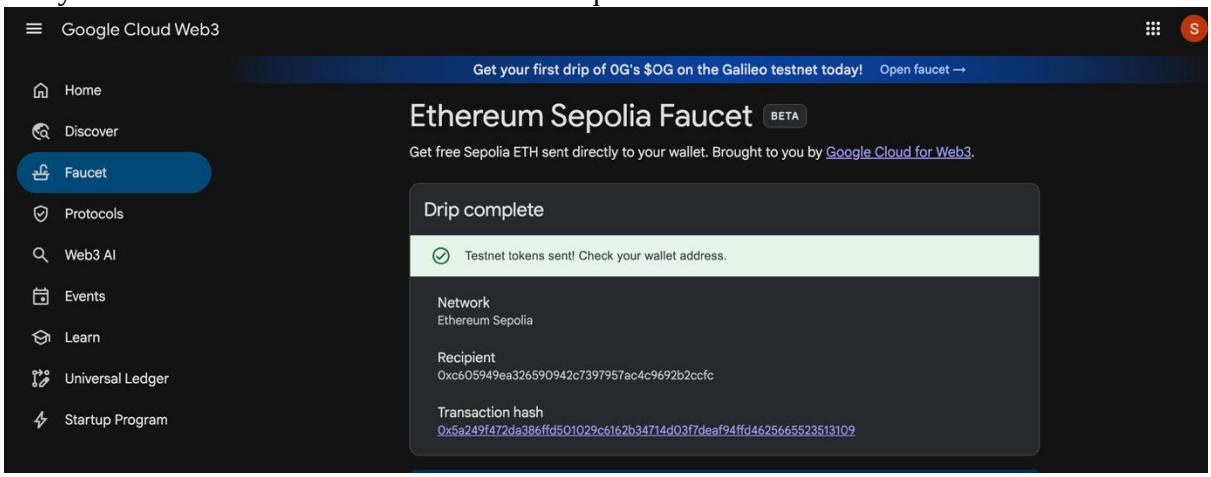


2. Secure Your Wallet:
   - During setup, you will be given a 12-word Secret Recovery Phrase.
   - WARNING: Write this phrase down on a physical piece of paper and store it somewhere safe. Anyone with this phrase can access your funds. Never share it with anyone.
   - Confirm the phrase to finish wallet creation.
3. Enable Test Networks:
   - In the MetaMask extension, click the network selection dropdown in the top-left corner.
   - Find the toggle or button that says "Show test networks" or "Show/hide test networks" and make sure it is On.

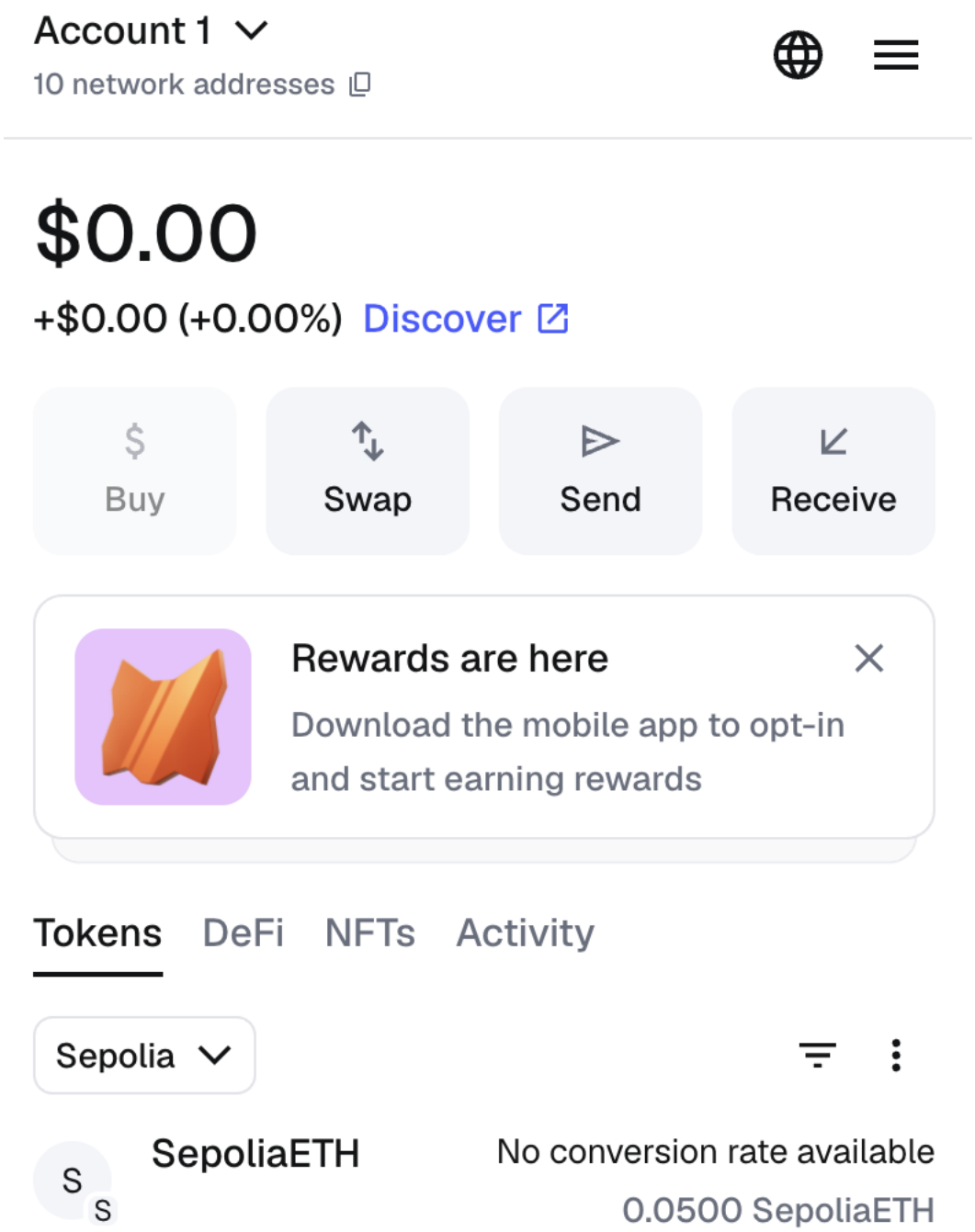
Part B: Acquire Test Funds

1. Select Sepolia Network:
   - Click the network dropdown again and select the "Sepolia" network from the list.

2. Copy Your Wallet Address:
   - At the top of the MetaMask window, click on your account name (e.g., "Account 1") to copy your public wallet address to the clipboard. It will look like 0x123...abc.
3. Use a Sepolia Faucet:
   - A "faucet" is a website that gives free test-network currency to developers.
   - Navigate to a Sepolia faucet in your browser. Common options include:
     - sepoliafaucet.com (Alchemy)
     - cloud.google.com/application/web3/faucet/ethereum/sepolia (Google Cloud)
   - You may need to sign up for a free account on these services.
   - Paste your wallet address into the faucet and request funds.



4. Verify Funds:
   - Wait 1-2 minutes. You will see a small amount of "SepoliaETH" (e.g., 0.05 ETH) appear in your MetaMask wallet. You are now ready to pay for test transactions.



-

Part C: Write the Smart Contract

1. Open Remix IDE:
   - Navigate to remix.ethereum.org in your browser.
   - Dismiss any introductory pop-ups.
2. Create Contract File:
   - In the "File Explorer" tab on the left, find the contracts folder.

- Right-click the contracts folder and select "New File."
- Name the new file SimpleStorage.sol and press Enter.

3. Enter Contract Code:
   - The file will open in the main editor.
   - Copy and paste the following code exactly into the file:

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

/**
 * @title SimpleStorage
 * @dev A basic contract that lets you store and retrieve a number.
 */
contract SimpleStorage {

    uint256 private myNumber;

    /**
     * @dev Stores a new number.
     * @param _newNumber The new number to store.
     */
    function store(uint256 _newNumber) public {
        myNumber = _newNumber;
    }

    /**
     * @dev Retrieves the stored number.
     * @return The currently stored number.
     */
    function retrieve() public view returns (uint256) {
        return myNumber;
    }
}
```

Part D: Compile the Contract

1. Navigate to Compiler:
   - On the far-left icon bar, click the "Solidity Compiler" tab.

2. Set Compiler Version:
   - Ensure the "Compiler" version dropdown is set to 0.8.20 or any version higher in the 0.8.x range (which matches the pragma line in the code).

3. Compile:
   - Click the large blue "Compile SimpleStorage.sol" button.
   - Expected Result: A green checkmark will appear on the "Solidity Compiler" icon, indicating a successful compilation.

Part E: Deploy to Sepolia Testnet

1. Navigate to Deploy Tab:
   - On the far-left icon bar, click the "Deploy & Run Transactions" tab.

2. Connect MetaMask:

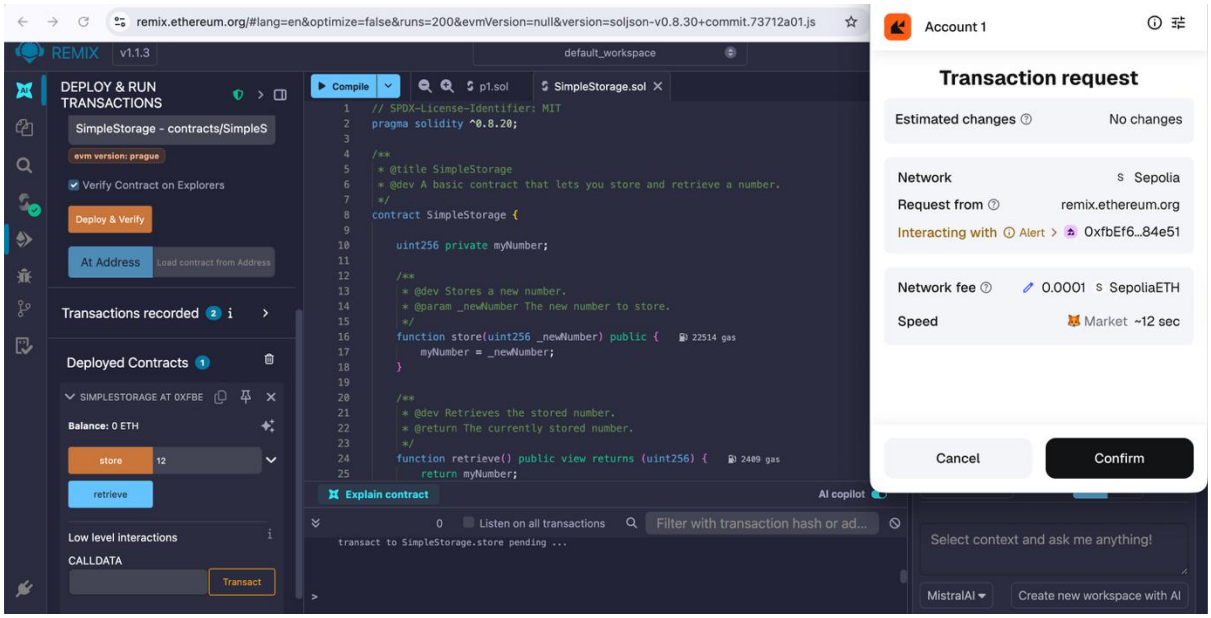
- At the top of the deploy panel, click the "ENVIRONMENT" dropdown.
- Select "Injected Provider - MetaMask".
- A MetaMask pop-up will appear. Click "Next" and then "Connect" to authorize Remix.
- Verification: The "ENVIRONMENT" box will now show "Sepolia (11155111)" and your wallet address will be listed under "ACCOUNT".

3. Deploy the Contract:
   - In the "CONTRACT" dropdown, ensure SimpleStorage - contracts/SimpleStorage.sol is selected.
   - Click the orange "Deploy" button.
   - A new MetaMask pop-up will appear, asking you to confirm the transaction (this is where you spend your test ETH).
   - Scroll down and click "Confirm".
4. Verify Deployment:
   - Wait a few seconds. In the Remix terminal at the bottom, you will see a green checkmark and transaction details.
   - In the "Deploy" panel, scroll down to the "Deployed Contracts" section. You will see your SIMPLESTORAGE contract listed.

4. Verification: Test the Deployed Contract

You will now interact with the live contract on the testnet.

1. Expand Contract:
   - In the "Deployed Contracts" section, click the small triangle next to your SIMPLESTORAGE contract to expand its interface.
2. Test the retrieve Function (Read):
   - Click the blue retrieve button.
   - Result: The value 0 will appear below the button. This is the default value for the myNumber variable.
3. Test the store Function (Write):
   - In the text box next to the orange store button, type the number 42.
   - Click the store button.
   - A MetaMask pop-up will appear to confirm this new transaction (since you are changing data on the blockchain). Click "Confirm".
4. Wait and Verify:
   - Wait 5-10 seconds for the transaction to be confirmed on the Sepolia network.
   - Click the blue retrieve button again.
   - Final Result: The value 42 will now appear below the button.



5. Conclusion

Congratulations. You have successfully written, compiled, and deployed an interactive smart contract to a public Ethereum testnet. You have learned the fundamental workflow of a blockchain developer using the most common beginner tools.

EXPERIMENT 5

DESIGN AND DEVELOP CRYPTOCURRENCY FOR MULTIPLE USER USING PYTHON.

In this lab, you will set up a local, command-line interface (CLI) cryptocurrency wallet. You will simulate a blockchain environment to understand:

1. **Wallet Management:** How users are identified by addresses.
2. **Transactions:** How value moves between accounts using immutable records.
3. **Gas Fees:** The cost of processing transactions on a network.
4. **Blocks:** How transactions are grouped and mined.

Prerequisites

- Python 3.8 or higher installed.
- Basic familiarity with the Command Prompt/Terminal.

Part 1: Environment Setup

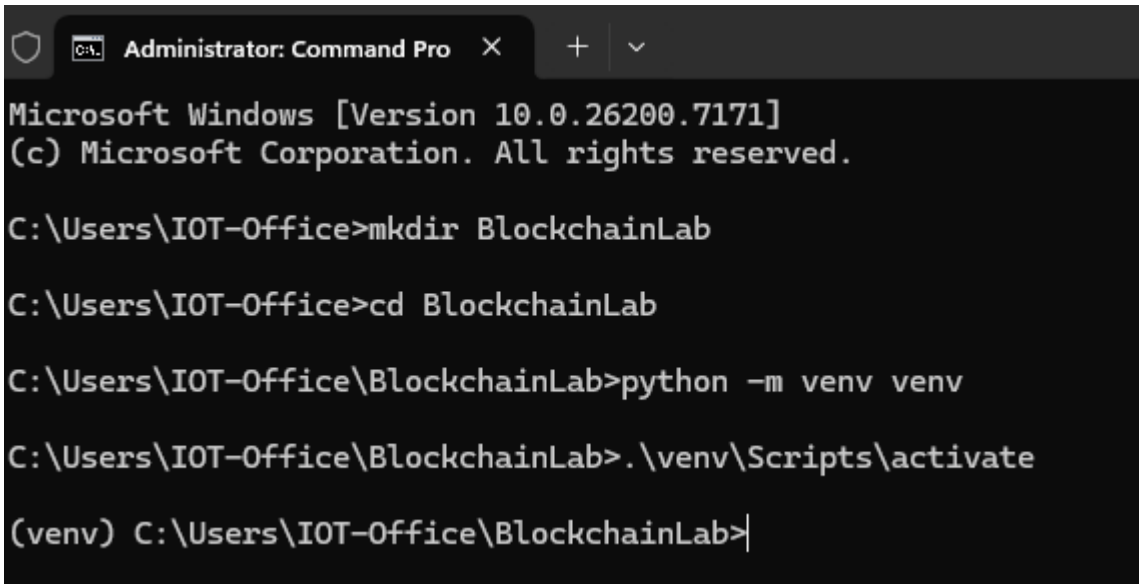
Before writing code, we need to create an isolated "Virtual Environment" (venv) to install our blockchain libraries without affecting your main system.

Step 1: Create the Virtual Environment

For Windows (PowerShell or CMD):

```
# 1. Create a new folder for your lab
mkdir BlockchainLab
cd BlockchainLab

# 2. Create the virtual environment named 'venv'
python -m venv venv

# 3. Activate the environment
.\venv\Scripts\activate
```

You will know it worked if you see (venv) appear at the start of your command line.



For Linux / macOS (Terminal):

```
# 1. Create a new folder
mkdir BlockchainLab
cd BlockchainLab

# 2. Create the virtual environment
python3 -m venv venv

# 3. Activate the environment
source venv/bin/activate
```

Step 2: Install Dependencies

We need the web3.py library. We specifically need the [tester] extra, which includes a simulated blockchain that runs in your computer's RAM (Random Access Memory).

Type the following command exactly (include the quotes):

```
pip install "web3[tester]"
```

```
(venv) C:\Users\IOT-Office\BlockchainLab>pip install "web3[tester]"
Collecting web3[tester]
  Downloading web3-7.14.0-py3-none-any.whl.metadata (5.6 kB)
Collecting eth-abi>=5.0.1 (from web3[tester])
  Downloading eth_abi-5.2.0-py3-none-any.whl.metadata (3.8 kB)
Collecting eth-account>=0.13.6 (from web3[tester])
  Downloading eth_account-0.13.7-py3-none-any.whl.metadata (3.7 kB)
Collecting eth-hash>=0.5.1 (from eth-hash[pycryptodome]>=0.5.1->web3[tester])
  Downloading eth_hash-0.7.1-py3-none-any.whl.metadata (4.2 kB)
Collecting eth-typing>=5.0.0 (from web3[tester])
  Downloading eth_typing-5.2.1-py3-none-any.whl.metadata (3.2 kB)
```

```
Successfully installed aiohappyeyeballs-2.6.1 aiohttp-3.13.2 aiosignal
-3.8.0 cached-property-2.0.1 certifi-2025.11.12 charset_normalizer-3.4
unt-0.13.7 eth-bloom-3.1.0 eth-hash-0.7.1 eth-keyfile-0.8.1 eth-keys-0
5.2.1 eth-utils-5.3.1 frozenlist-1.8.0 hexbytes-1.3.1 idna-3.11 lru-di
ache-0.4.1 py-ecc-8.0.0 py-evm-0.12.1b1 py-geth-6.3.0 pycryptodome-3.2
ize-17.0.0 pywin32-311 regex-2025.11.3 requests-2.32.5 rlp-4.1.0 safe-
ers-2.4.0 toolz-1.1.0 trie-3.1.0 types-requests-2.32.4.20250913 typing
-2.5.0 web3-7.14.0 websockets-15.0.1 yarl-1.22.0

[notice] A new release of pip is available: 25.2 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip
```

```
(venv) C:\Users\IOT-Office\BlockchainLab>python.exe -m pip install --upgrade pip
Requirement already satisfied: pip in c:\users\iot-office\blockchainlab\venv\lib\site-packages (25.2)
Collecting pip
  Downloading pip-25.3-py3-none-any.whl.metadata (4.7 kB)
Downloading pip-25.3-py3-none-any.whl (1.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.8/1.8 MB 31.9 MB/s  0:00:00
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 25.2
    Uninstalling pip-25.2:
      Successfully uninstalled pip-25.2
Successfully installed pip-25.3
```

Part 2: The Code

Create a new file named wallet_shell.py inside your folder and paste the following code.

```
(venv) C:\Users\IOT-Office\BlockchainLab>notepad wallet_shell.py

(venv) C:\Users\IOT-Office\BlockchainLab>
```

File: wallet_shell.py

```
import sys
import cmd
from web3 import Web3

class WalletShell(cmd.Cmd):
    intro = 'Welcome to the ClassCrypto Shell. Type help or ? to list commands.\n'
    prompt = '(crypto) > '
```

```
def __init__(self):
    super().__init__()
    # 1. Initialize the Ghost Blockchain
    # EthereumTesterProvider creates a temporary chain in your RAM.
    # It handles mining, signatures, and gas automatically.
    self.w3 = Web3(Web3.EthereumTesterProvider())

    # 2. Setup "Users"
    # We map friendly names (Alice) to the complex Hex addresses (0x7A...)
    # provided by the test environment.
    self.accounts = self.w3.eth.accounts
    self.users = {
        "bank": self.accounts[0],
        "alice": self.accounts[1],
        "bob": self.accounts[2],
        "charlie": self.accounts[3]
    }
    print(f"Blockchain initialized with {len(self.users)} users: Bank, Alice, Bob, Charlie")

def do_balance(self, arg):
    """
    Check balance of a user.
    Usage: balance <user_name>
    Example: balance alice
    """
    name = arg.lower().strip()
    if name not in self.users:
        print(f"Error: User '{name}' not found. Available: {list(self.users.keys())}")
        return

    address = self.users[name]
    # Balances are returned in 'Wei' (tiny units), we convert to 'Ether' for readability
    wei_balance = self.w3.eth.get_balance(address)
    eth_balance = self.w3.from_wei(wei_balance, 'ether')

    print(f"User: {name.capitalize()}")
    print(f"Address: {address}")
    print(f"Balance: {eth_balance} ETH")

def do_send(self, arg):
    """
    Send crypto between users.
    Usage: send <from> <to> <amount>
    Example: send bank alice 10
    """
    args = arg.split()
    if len(args) != 3:
        print("Error: Usage is 'send <from> <to> <amount>'")
        return

    sender, receiver, amount = args[0].lower(), args[1].lower(), args[2]

    if sender not in self.users or receiver not in self.users:
        print("Error: Unknown user. Please use: bank, alice, bob, charlie")
```

```
            return

        try:
            amount_eth = float(amount)
            sender_addr = self.users[sender]
            receiver_addr = self.users[receiver]

            print(f"Processing transaction: {sender.capitalize()} -> {receiver.capitalize()} ({amount_eth}
ETH)...")

            # 1. Submit the Transaction
            # In a real app, you would need the Private Key to sign this.
            # The TesterProvider handles the signing automatically for us.
            tx_hash = self.w3.eth.send_transaction({
                'from': sender_addr,
                'to': receiver_addr,
                'value': self.w3.to_wei(amount_eth, 'ether')
            })

            # 2. Wait for Mining (Validation)
            # The network needs time to include the transaction in a block.
            receipt = self.w3.eth.wait_for_transaction_receipt(tx_hash)

            print(f"Success! Block #{receipt.blockNumber} mined.")
            print(f"Gas Used: {receipt.gasUsed} (Cost to process)")
            print(f"Tx Hash: {tx_hash.hex()}")

        except ValueError as e:
            print(f"Transaction Failed: {e}")
        except Exception as e:
            print(f"System Error: {e}")

    def do_chain(self, arg):
        """
        Inspect the latest block in the chain.
        """
        latest = self.w3.eth.get_block('latest')
        print("\n--- LATEST BLOCK ---")
        print(f"Block Number: {latest.number}")
        print(f"Block Hash:   {latest.hash.hex()}")
        print(f"Transactions: {len(latest.transactions)}")
        print("--------------------")

    def do_exit(self, arg):
        """Exit the shell."""
        print("Closing connection...")
        return True

if __name__ == '__main__':
    try:
        WalletShell().cmdloop()
    except KeyboardInterrupt:
        print("\nExiting...")
```

Part 3: Understanding the Tools

1. Web3.py (The Bridge)

In the real world, a blockchain (like Ethereum) is a network of thousands of computers. Your Python script isn't part of that network; it needs a bridge to talk to it. **Web3.py** is that bridge. It translates your Python commands into the specific JSON-RPC language that blockchain nodes understand.

2. EthereumTesterProvider (The Simulation)

Normally, you would connect to a real node (using HTTPProvider). However, for this lab, we use EthereumTesterProvider. This creates a **Ghost Blockchain** entirely inside your RAM. It validates signatures, mines blocks, and checks balances exactly like the real thing, but it disappears when you close the script.

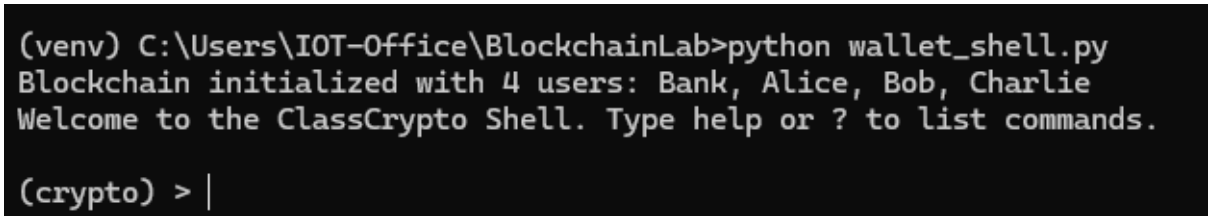
3. The cmd Module

This is a standard Python library that allows us to create interactive shells easily. It handles the (crypto) > prompt and parsing your commands so you don't have to write while True: input() loops manually.

Part 4: Lab Walkthrough

Step 1: Run the Shell
Make sure your venv is active and run:
python wallet_shell.py

You should see the welcome message and the prompt (crypto) >.



Step 2: Check the "Bank"
The simulation starts with pre-funded accounts. Let's see how much money the "Bank" has.
(crypto) > balance bank

Record the Hex Address and the Balance.

Step 3: Check "Alice"
Alice should start with 1,000,000 ETH as well (in this test environment).
(crypto) > balance alice

```
(crypto) > balance Alice
User: Alice
Address: 0x2B5AD5c4795c026514f8317c7a215E218DcCD6cF
Balance: 1000000 ETH
(crypto) >
```

Step 4: Execute a Transaction
Let's move funds. This simulates:

1. Creating a transaction payload.
2. Signing it (handled auto-magically by the tester).
3. Broadcasting it to the network.
4. Mining it into a block.

(crypto) > send bank alice 50.5

Observe the output. Note the "Gas Used". This is the fee paid to the network to process the data.

```
(crypto) > send bank alice 50.5
Processing transaction: Bank -> Alice (50.5 ETH)...
C:\Users\IOT-Office\BlockchainLab\venv\Lib\site-packages\cached_property.py:27: DeprecationWarning: '
function' is deprecated and slated for removal in Python 3.16; use inspect.iscoroutinefunction() inst
  if asyncio.iscoroutinefunction(self.func):
Success! Block #1 mined.
Gas Used: 21000 (Cost to process)
Tx Hash: d5914dc1131b239f9dc25ed11f74fa877348325233dd89ea0375ade118e912d7
```

Step 5: Verify the Ledger
Check the balances again.
(crypto) > balance bank
(crypto) > balance alice

```
(crypto) > balance bank
User: Bank
Address: 0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf
Balance: 999949.499979 ETH
(crypto) >
```

```
User: Alice
Address: 0x2B5AD5c4795c026514f8317c7a215E218DcCD6cF
Balance: 1000050.5 ETH
(crypto) >
```

Question: Did the Bank lose exactly 50.5 ETH? Or did they lose 50.5 plus a tiny bit more? (Hint: Gas Fees).

Step 6: Inspect the Chain
Look at the block that was just mined containing your transaction.
(crypto) > chain

```
(crypto) > chain

--- LATEST BLOCK ---
Block Number: 1
Block Hash:   4c934b01e84c7f24e3eb48d8c521b68dbc954a023ef617222501ccd9a8c0f648
Transactions: 1
--------------------
(crypto) >
```

Step 7: The "Double Spend" (Optional)
Try to send more money than Alice has.
(crypto) > send alice bob 2000000

```
(crypto) > send Alice Bob 2000000
Processing transaction: Alice -> Bob (2000000.0 ETH)...
System Error: Sender does not have enough balance to cover transaction value and gas  (has 1000050500000000000000000, ne
eds 2000000000021000000000000)
(crypto) >
```

Observe the Python error. The blockchain logic rejects the invalid state transition.

EXPERIMENT 7

Interacting with the Balance Transfer Smart Contract in Remix

Objective: To understand, compile, deploy, and test the functionality of the Balance Transfer contract. This includes funding it at deployment, sending it Ether after deployment, and successfully (and unsuccessfully) using the transfer function.

Tools:

- Remix IDE (a web-based Solidity IDE)

Part 1: Understanding the Smart Contract

First, let's look at the BalanceTransfer.sol contract we will be using. This contract is designed to do one simple thing: receive Ether from anyone, but only allow its owner to withdraw that Ether to a specified address.

The Code

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

/**
 * @title BalanceTransfer
 * @dev A simple contract that can receive Ether and allows only the
 * owner to transfer that Ether out to another address.
 */
contract BalanceTransfer {
    // Variable to store the address of the contract deployer.
    address public owner;

    // The constructor is called only once, when the contract is deployed.
    // 'payable' allows it to receive Ether during deployment.
    constructor() payable {
        // 'msg.sender' is the address that deployed the contract.
        // We set this address as the 'owner'.
        owner = msg.sender;
    }

    // This is the receive() fallback function.
    // It's 'external' (callable from outside) and 'payable' (can receive Ether).
    // This function is automatically triggered when someone sends Ether
    // to the contract without specifying any function to call.
    receive() external payable {}

    // This is the fallback() function.
    // It's also 'external' and 'payable'.
    // This triggers if someone sends Ether and data to the contract,
    // but the data doesn't match any other function name.
    fallback() external payable {}

    /**
     * @dev Transfers a specific amount of the contract's Ether balance
     * to a target address.
     * @param _to The address to send Ether to. Must be 'payable'.
```

```
     * @param _amount The amount of Ether (in Wei) to send.
     */
    function transfer(address payable _to, uint256 _amount) public {
        // Security Check 1: Only the 'owner' can call this function.
        // 'require' checks a condition. If it's false, the transaction
        // reverts (fails) and returns the error message.
        require(msg.sender == owner, "Only Contract owner can transfer balance");

        // Security Check 2: Ensure the contract has enough Ether to send.
        // 'address(this).balance' is the contract's current Ether balance.
        require(address(this).balance >= _amount, "Insufficient balance in contract");

        // If both checks pass, perform the transfer.
        // '_to.transfer(_amount)' sends the specified amount to the target address.
        _to.transfer(_amount);
    }

    /**
     * @dev A public 'view' function to check the contract's current balance.
     * 'view' means it only reads data and doesn't cost any gas to call
     * when called externally (not from another contract).
     * @return The contract's balance in Wei.
     */
    function getContractBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

Code Explanation

1. **pragma solidity ^0.8.0;**: This line specifies that the code is written for Solidity version 0.8.0 or any newer version in that series.
2. **contract BalanceTransfer { ... }**: This defines the start of our smart contract named BalanceTransfer.
3. **address public owner;**: This declares a state variable named owner of type address. public means Solidity automatically creates a "getter" function for it, so you can check its value from outside the contract (as we'll do in the lab).
4. **constructor() payable { ... }**: This is the **constructor**. It runs *only once* when the contract is first deployed.
    - payable: This keyword is very important. It allows the constructor to receive Ether. This is how we will fund the contract *at the moment of deployment*.
    - owner = msg.sender;: msg.sender is a global variable that holds the address of the account *calling* the function. In a constructor, it's the address deploying the contract. We save this address in our owner variable.
5. **receive() external payable {}**: This is a special function. It's automatically executed whenever someone sends Ether to the contract's address *without* calling any specific function (or sending empty data). The payable keyword allows it to accept the Ether.
6. **fallback() external payable {}**: This is another special function, similar to receive(). It's a "catch-all" that runs if someone calls a function that doesn't exist on the contract. We've also made it payable to accept Ether in these cases.
7. **function transfer(...) public { ... }**: This is our main withdrawal function.
    - public: It can be called by anyone (though our require check will limit it).
    - require(msg.sender == owner, ...): This is our first security check. It ensures that the person calling this function (msg.sender) is the same one stored in our owner variable. If not, the transaction fails with the error message.

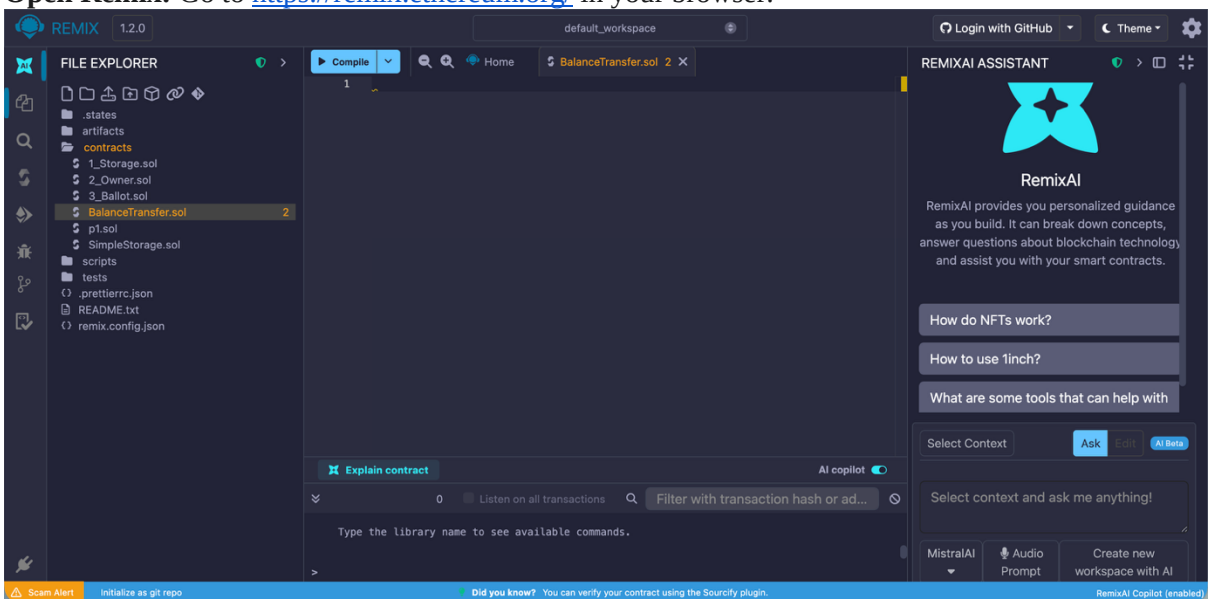
- require(address(this).balance >= _amount, ...): This is our second security check. It ensures the contract has enough funds to cover the requested transfer. address(this).balance is the contract's current Ether balance.
- _to.transfer(_amount);: If both checks pass, this line performs the actual Ether transfer to the recipient's address.

8. **function getContractBalance() public view returns (uint256) { ... }**: This is a simple helper function.
   - view: This means the function only *reads* data from the blockchain; it doesn't *change* any state. This makes it free to call (it doesn't cost any gas).
   - It returns the contract's current balance (address(this).balance) as a uint256 (unsigned integer).

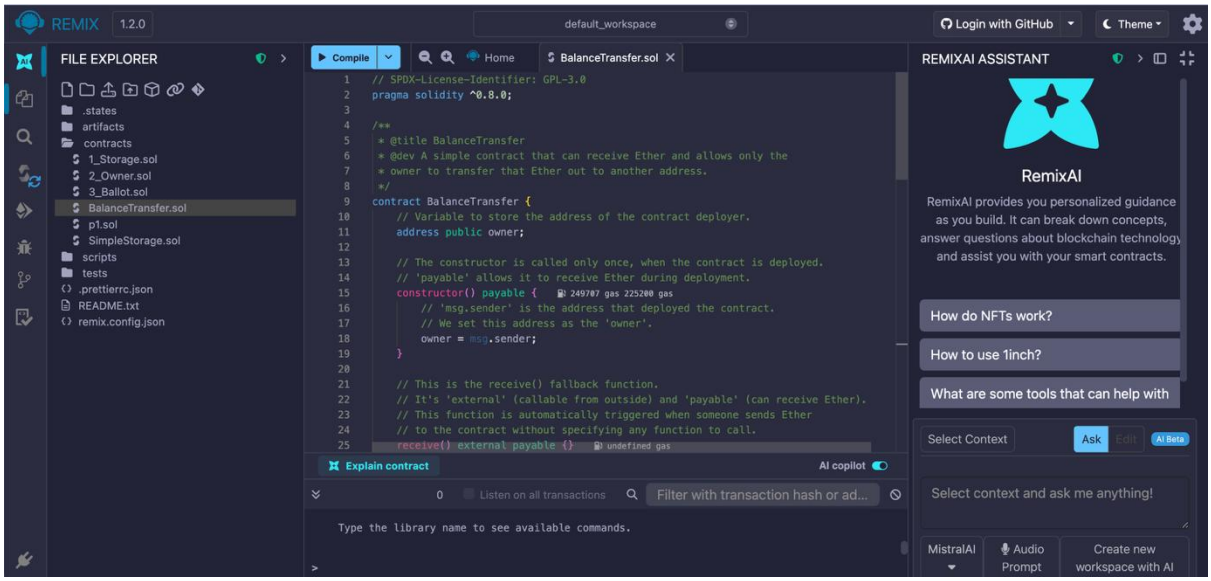
Now that we understand what the code does, let's compile and test it.

Part 2: Setup & Compilation

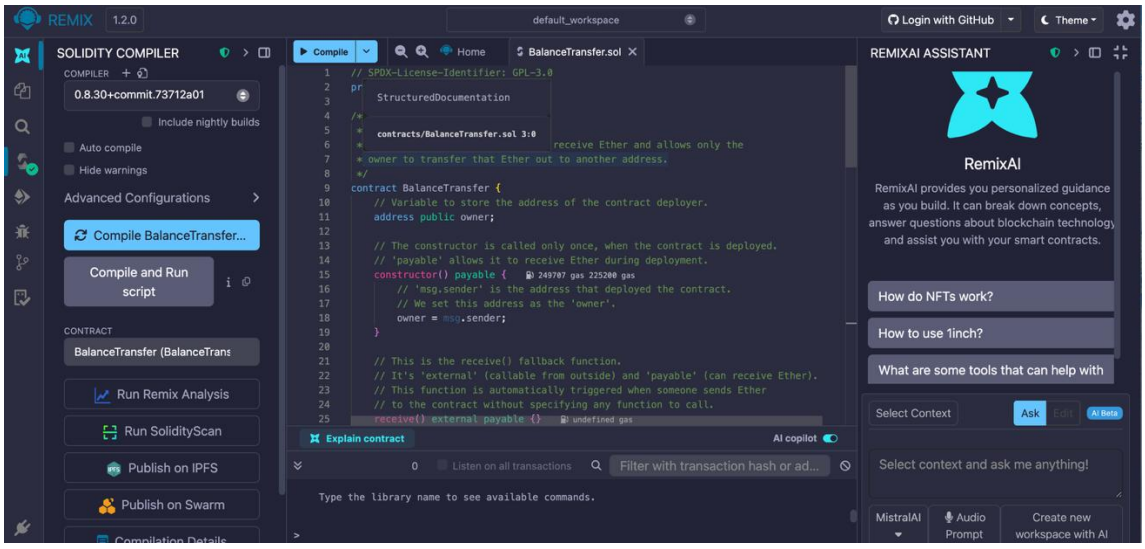
1. **Open Remix:** Go to https://remix.ethereum.org/ in your browser.



2. **Create Your File:**
   - In the "File Explorers" tab on the left, right-click on the contracts folder.
   - Select "New File".
   - Name the file BalanceTransfer.sol.
3. **Paste the Code:** Copy and paste the complete Solidity code from **Part 1** into the new BalanceTransfer.sol file editor.



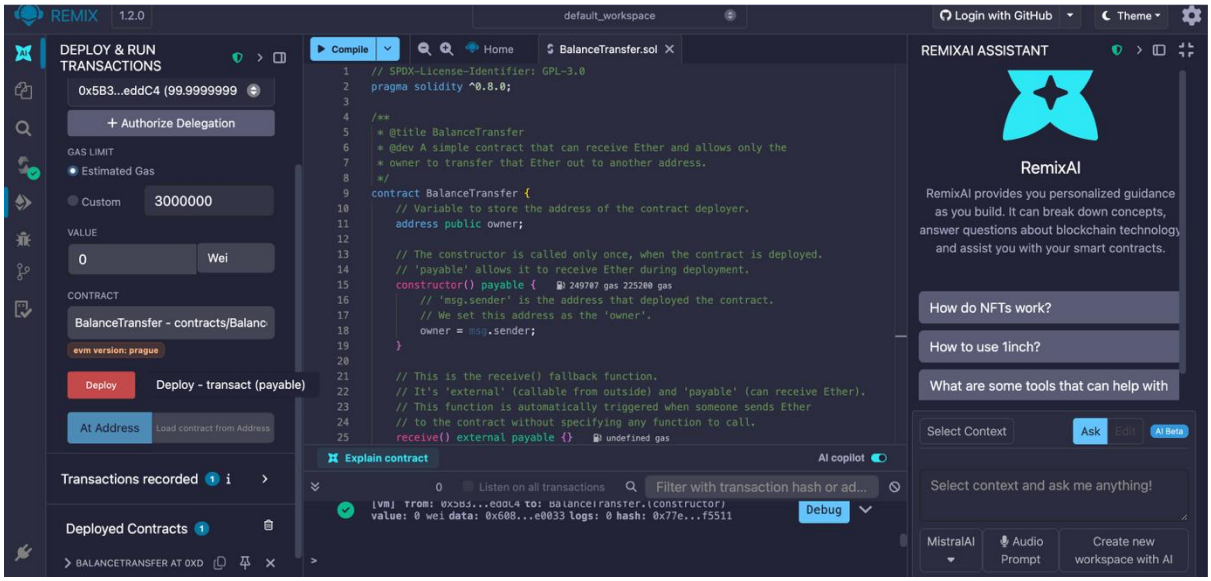
4. **Compile the Contract:**
   - Navigate to the "Solidity Compiler" tab (the third icon from the top on the far left).
   - Ensure the "Compiler" version selected (e.g., 0.8.20+commit...) is compatible with your code (^0.8.0). Any 0.8.x version will work.
   - Click the big blue **"Compile BalanceTransfer.sol"** button.
   - You should see a green checkmark in the compiler tab, indicating a successful compilation.



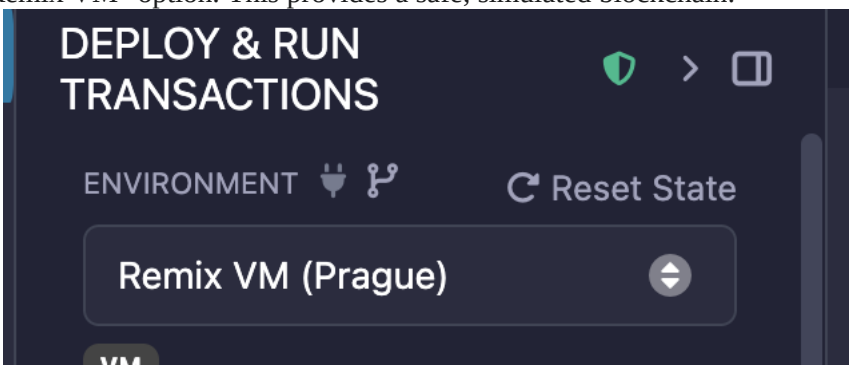
Part 3: Deployment & Initial Funding

Your contract's constructor is payable, which means you can send Ether to the contract *at the moment it's created*.

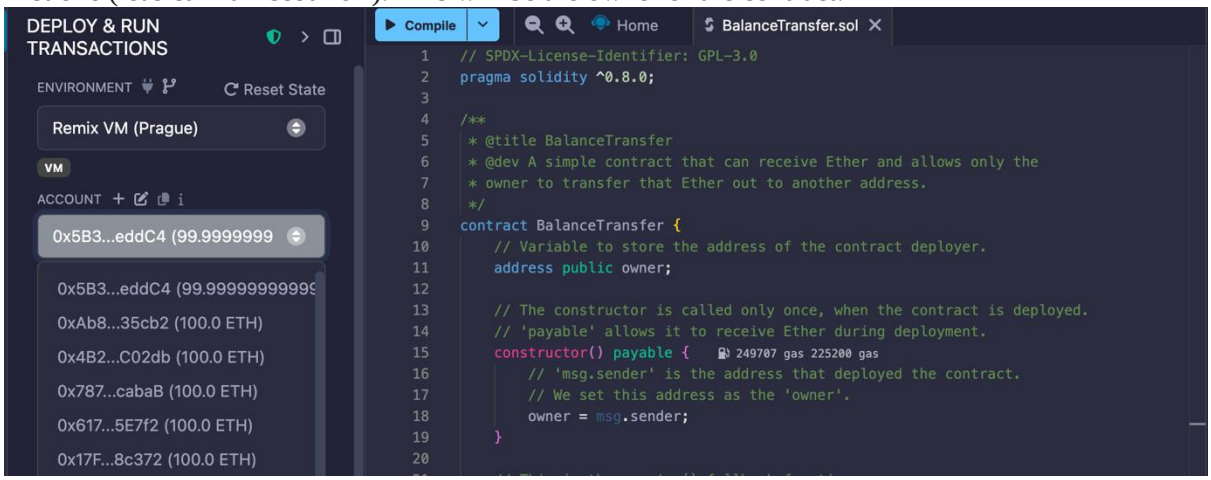
1. **Go to Deploy Tab:** Navigate to the "Deploy & Run Transactions" tab (the icon below the compiler).



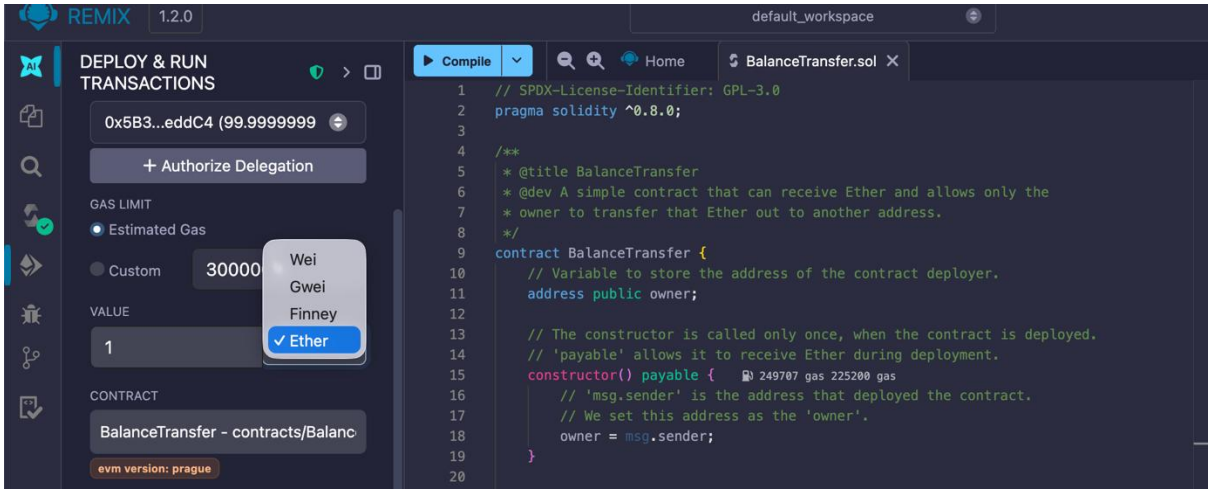
2. **Select Environment:** At the top, ensure the "Environment" is set to **"Remix VM (London)"** or any other "Remix VM" option. This provides a safe, simulated blockchain.



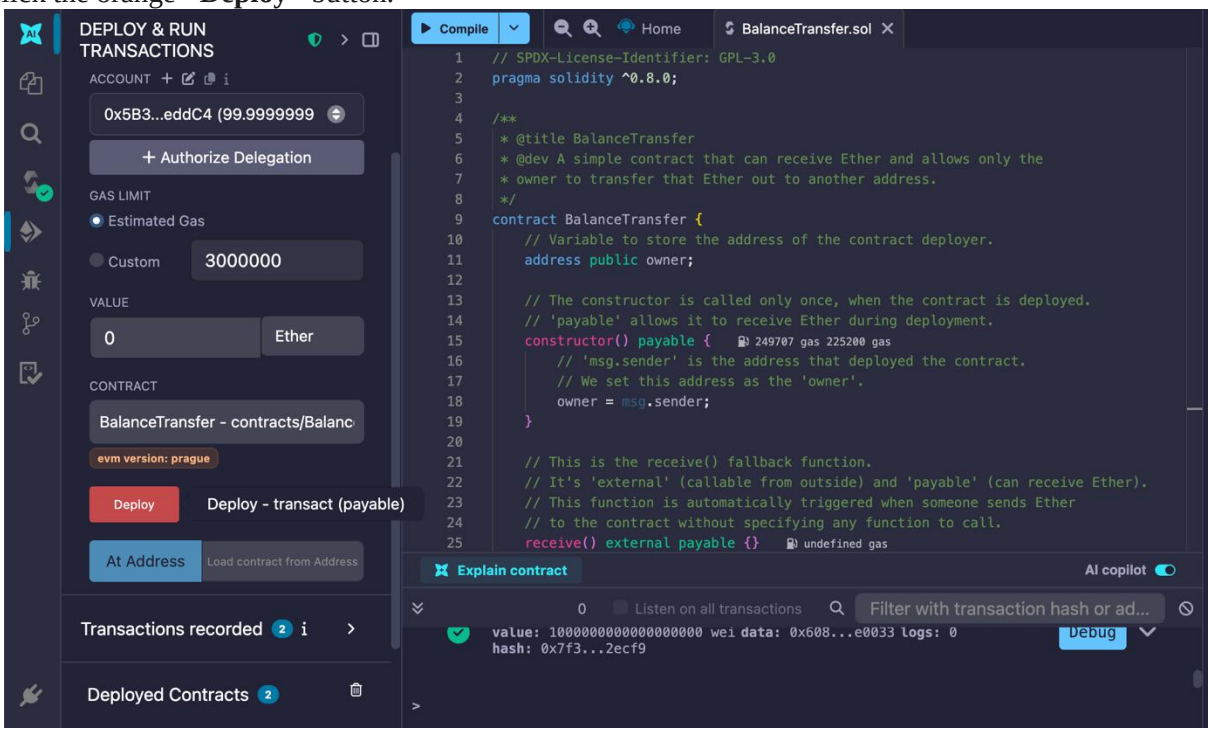
3. **Select Account:** The "ACCOUNT" dropdown shows your simulated accounts. Keep it on the first one (let's call it **Account 1**). This will be the owner of the contract.



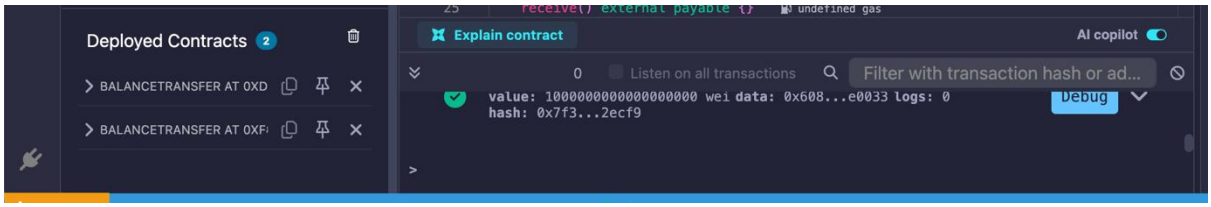
4. **Set Deployment Value:** This is the crucial step for funding.
   - Look for the **"VALUE"** field.
   - Enter 1 in the box.
   - Change the unit from "WEI" to **"ETHER"**.
   - This will send 1 Ether to the contract's constructor when you deploy.



5. **Deploy:**
   - Ensure your BalanceTransfer contract is selected in the "CONTRACT" dropdown.
   - Click the orange **"Deploy"** button.



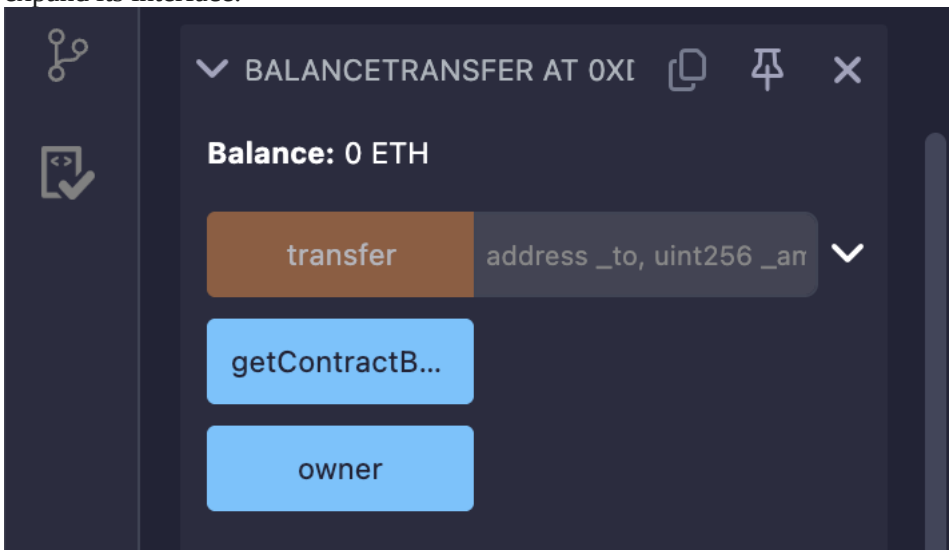
6. **Verify Deployment:**
   - Look at the Remix console (at the bottom). You'll see a log for the successful contract creation.
   - In the "Deploy & Run" tab, you'll now see your contract listed under "Deployed Contracts".



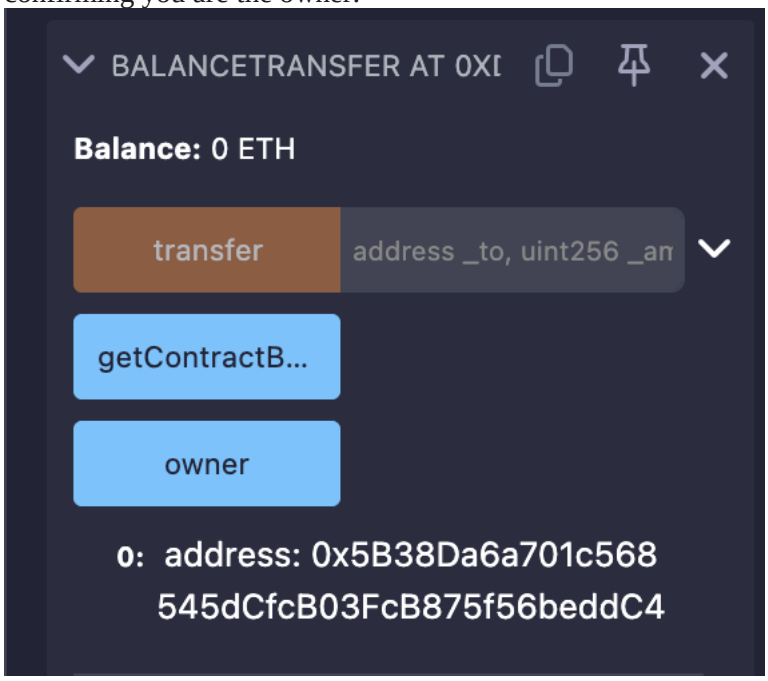
Part 4: Reading Contract State

Let's check the contract's initial state.

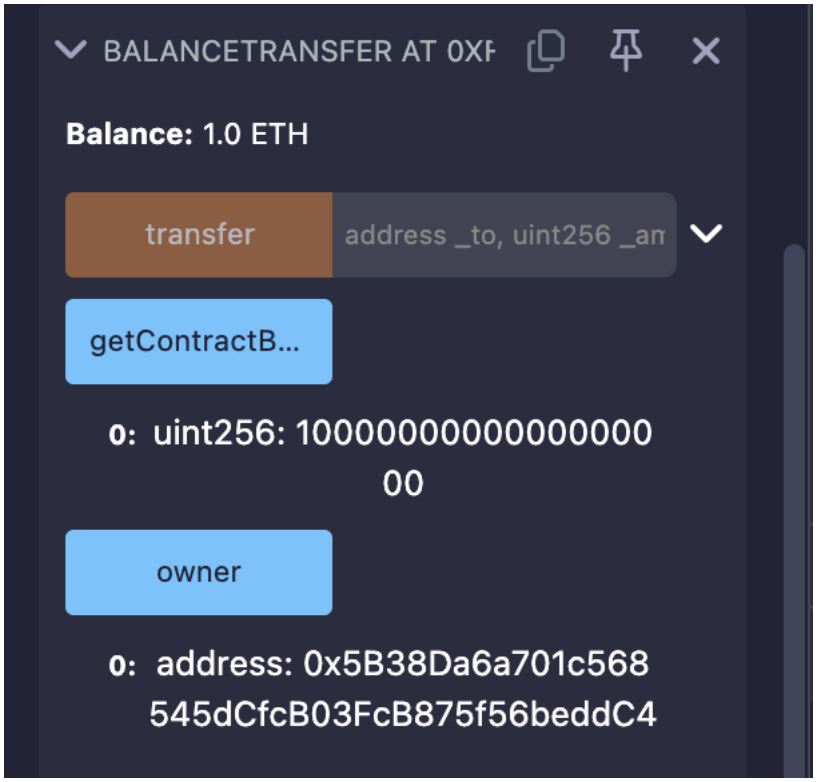
1. **Expand Contract:** Click the small arrow next to your deployed BalanceTransfer contract to expand its interface.



2. **Check Owner:** Click the blue owner button. It will display the address of **Account 1**, confirming you are the owner.



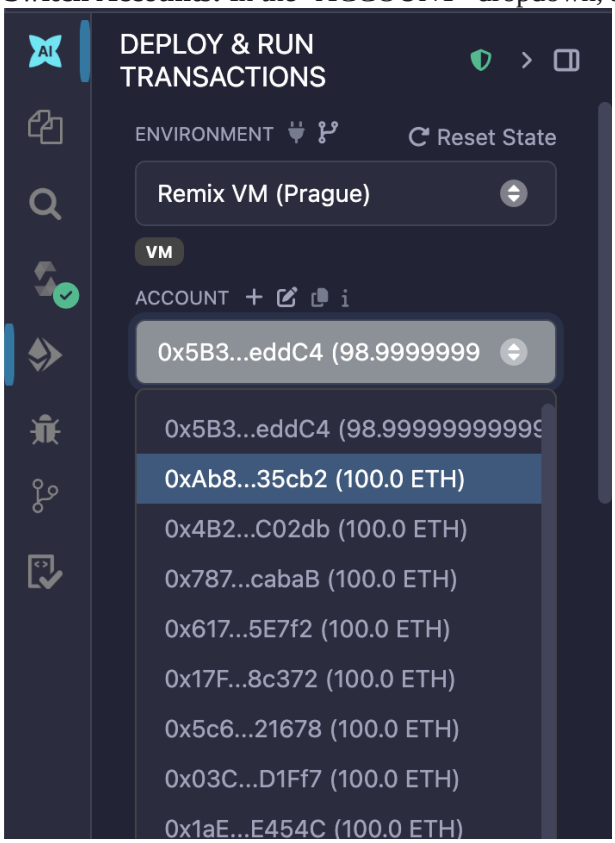
3. **Check Balance:** Click the blue getContractBalance button.
   - **Result:** It will return 1000000000000000000.
   - **Explanation:** This is 1 ETH expressed in **Wei** (the smallest denomination of Ether). 1 ETH = 1×10^{18} Wei. This confirms the 1 ETH you sent at deployment arrived safely.



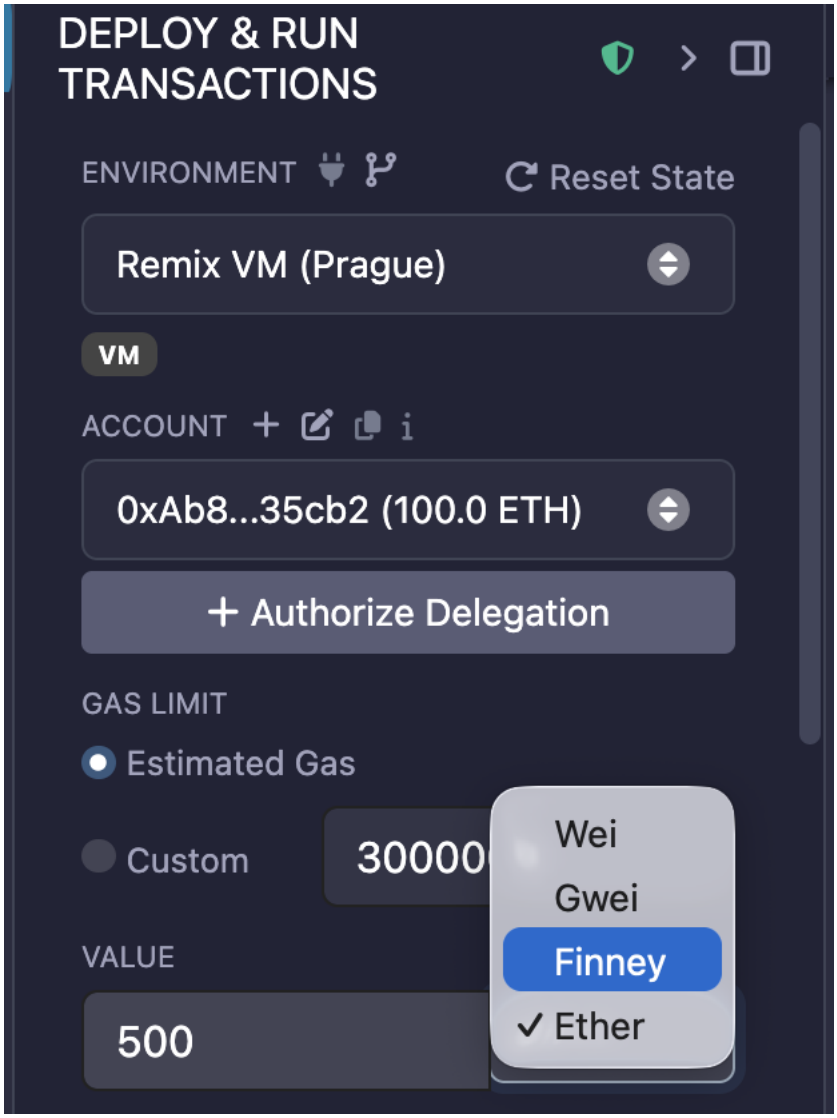
Part 5: Sending Ether to the Contract (After Deployment)

Your contract has a receive() function, allowing it to accept Ether sent to it *after* deployment.

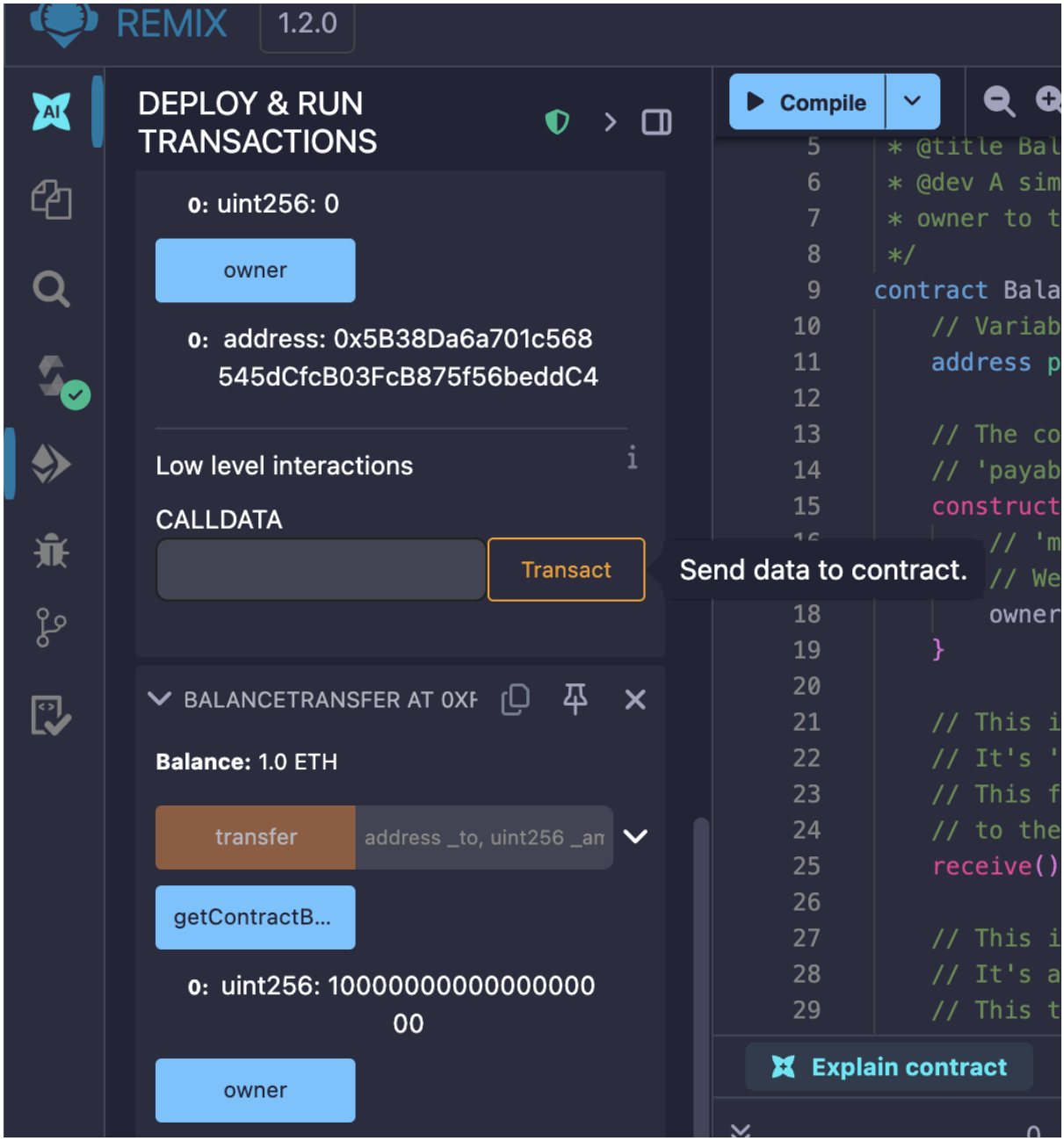
1. **Switch Accounts:** In the "ACCOUNT" dropdown, switch from **Account 1** to **Account 2**.



2. **Set Value:** In the "VALUE" field (at the top), enter 500 and set the unit to **"FINNEY"**. (1 ETHER = 1000 FINNEY, so this is 0.5 ETH).



3. **Send the Transaction:**
   - Find the "Low level interactions" section at the very bottom of your deployed contract's interface.
   - Leave the "Calldata" field empty.
   - Click the orange **"Transact"** button. This sends a transaction with 0.5 ETH and no function call data, which triggers your receive() function.



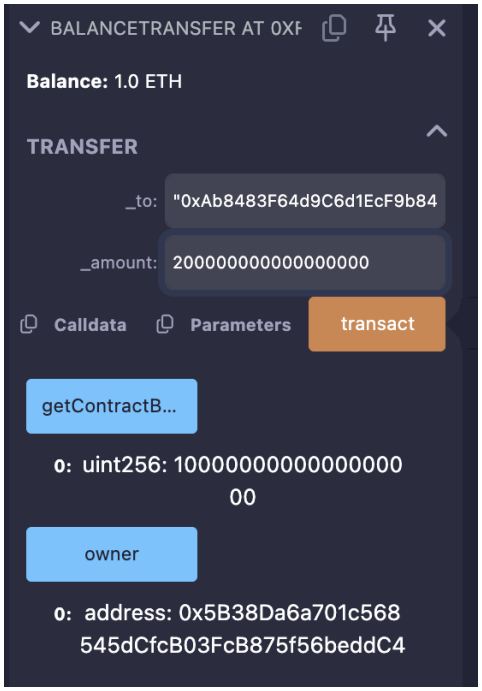
4. **Verify New Balance:**
   - Click getContractBalance again (you don't need to switch accounts for this, as it's a view function).
   - **Result:** It will now return 1500000000000000000 (1.5 ETH in Wei).

Part 6: Testing the Transfer Function (Success Case)

Now, let's transfer some of that balance out *as the owner*.

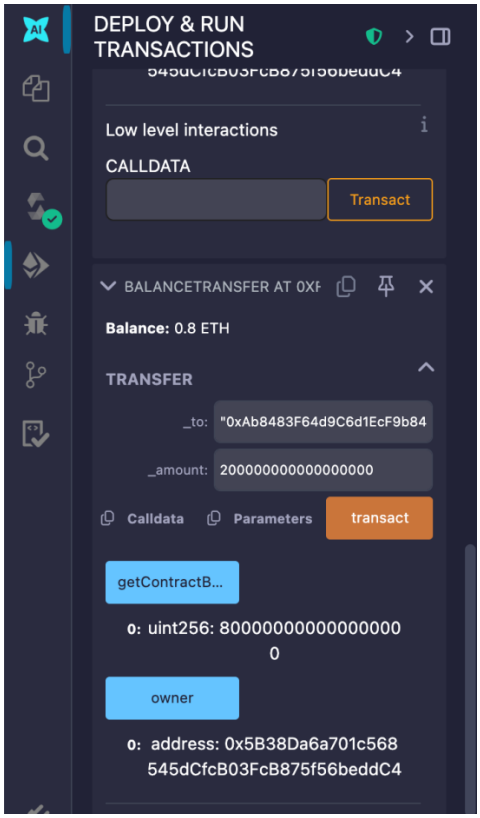
1. **Get Recipient Address:** In the "ACCOUNT" dropdown, select **Account 3**. Click the small "copy" icon next to the dropdown to copy its address.

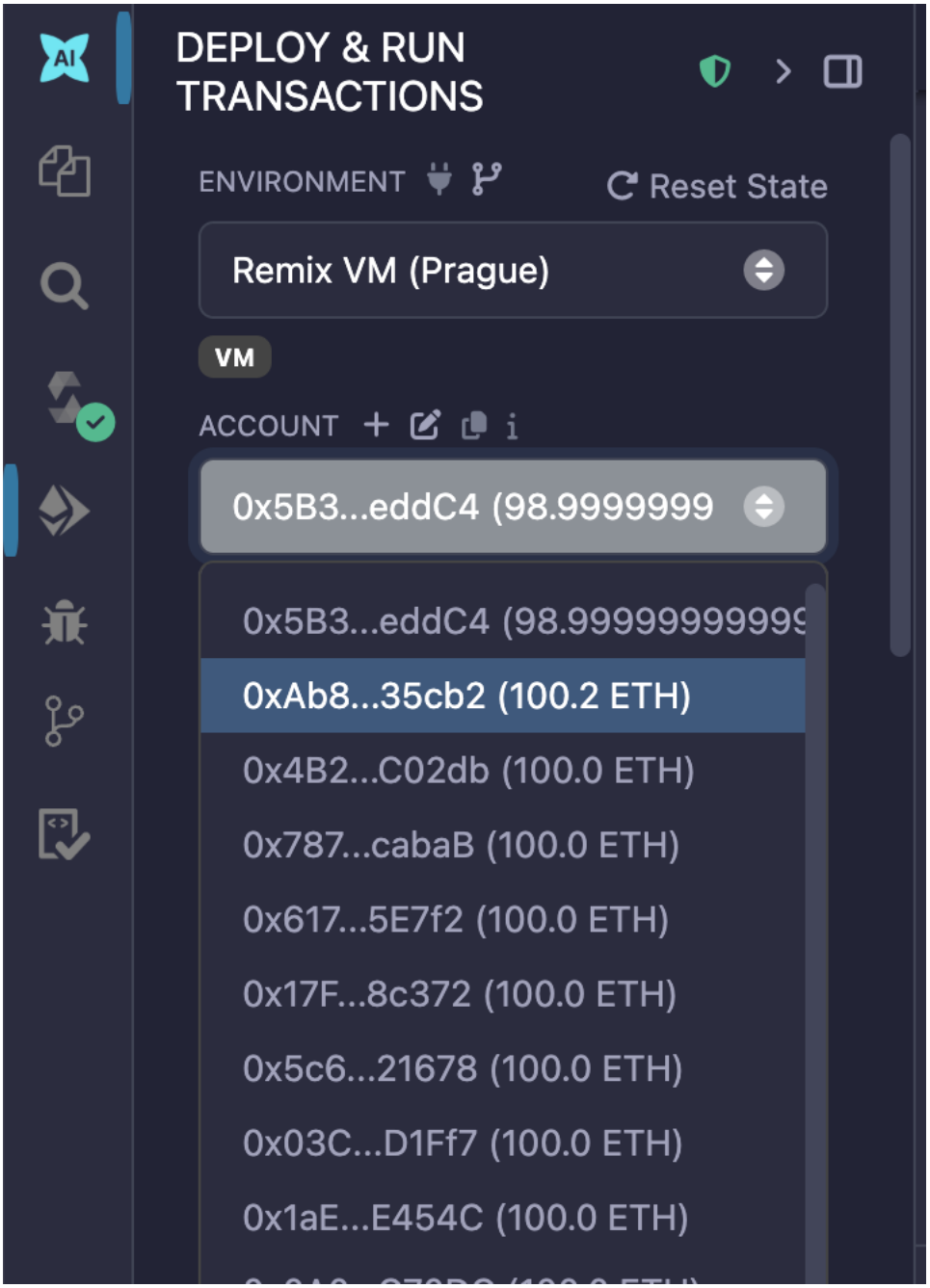
2. **Switch to Owner:** Switch the "ACCOUNT" dropdown back to **Account 1** (the owner).
3. **Enter Transfer Details:**
   - Find the transfer function in your deployed contract's interface.
   - In the _to field, paste the address for **Account 3**.
   - In the _amount field, enter the amount in **Wei**. Let's send 0.2 ETH. Type: 200000000000000000.



4. **Execute Transfer:** Click the orange transfer button.
5. **Verify Results:**
   - **Contract Balance:** Click getContractBalance. It should now show 1300000000000000000 (1.3 ETH).

     Recipient Balance: Check the balance of **Account 3** in the "ACCOUNT" dropdown.





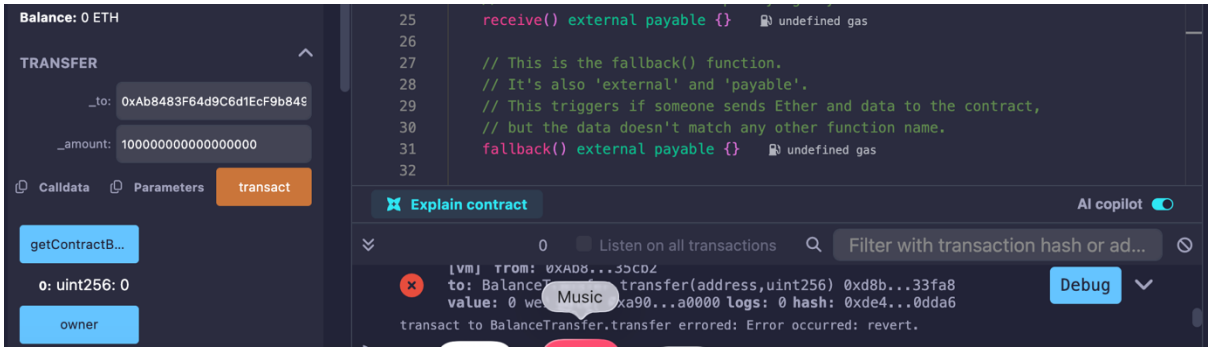
Part 7: Testing the require Fails (Failure Cases)

A good test also checks that your security measures work.

Failure Case 1: Not the Owner

1. **Switch Account:** In the "ACCOUNT" dropdown, switch to **Account 2** (a non-owner).
2. **Attempt Transfer:** Try to call the transfer function again with any amount (e.g., to Account 3, for 100 Wei).
3. **Observe Failure:** The transaction will fail. The Remix console will show an error:
   revert
   The transaction has been reverted to the initial state.
   Reason: Only Contract owner can transfer balance

This confirms your require(msg.sender == owner) check is working.



Failure Case 2: Insufficient Funds

1. **Switch to Owner:** Switch the "ACCOUNT" dropdown back to **Account 1**.
2. **Attempt to Overdraw:** The contract has 1.3 ETH. Let's try to transfer 2 ETH.
   - _to: Paste Account 3's address.
   - _amount: 2000000000000000000
3. **Observe Failure:** Click transfer. The transaction will fail. The console will show: revert

The transaction has been reverted to the initial state.
Reason: Insufficient balance in contract

This confirms your require(address(this).balance >= _amount) check is working.

Conclusion

You have successfully understood, compiled, deployed, and tested your BalanceTransfer contract. You have verified:

- How the constructor, receive, transfer, and view functions work.
- How to send Ether to a contract *during deployment*.
- How to read data from a contract (owner, getContractBalance).
- How to send Ether to a deployed contract's receive function.
- How to successfully call a protected function (transfer) *as the owner*.
- How your require statements correctly block unauthorized (Only Contract owner...) or invalid (Insufficient balance...) transactions.

EXPERIMENT 8

WRITE A PROGRAM TO PERFORM TOKEN CREATION AND MANAGEMENT ON ETHEREUM

Objective

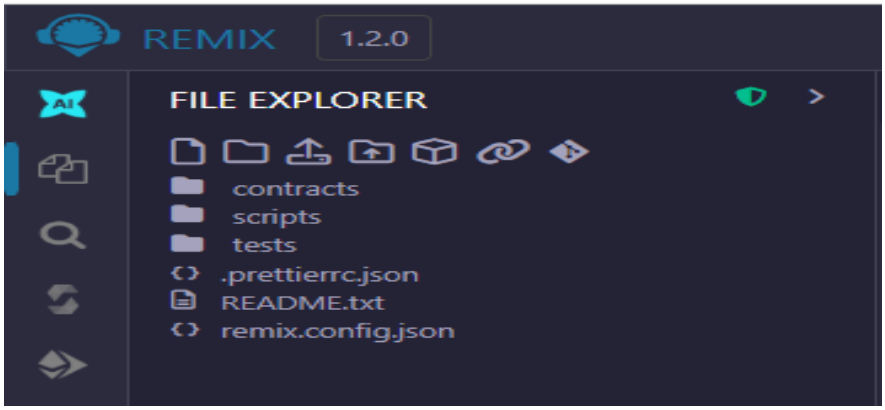
In this lab, you will create your own Cryptocurrency Token (ERC-20 style) and perform management operations like minting and transferring funds using the **Remix IDE**.

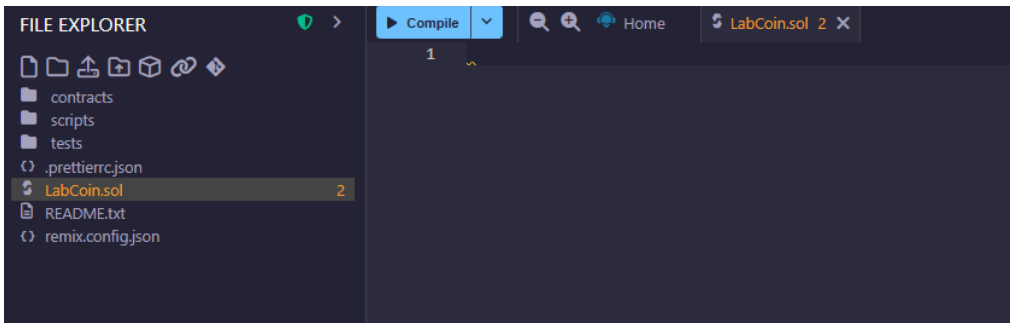
Prerequisites

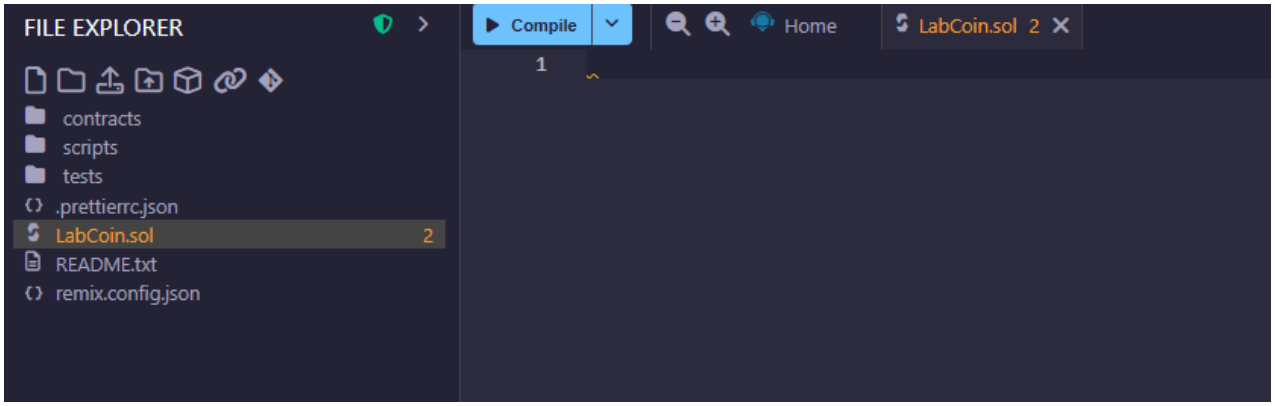
- A modern web browser (Chrome, Firefox, Brave).
- No installation required.

Part 1: The Setup

1. Open your browser and navigate to **remix.ethereum.org**.
2. On the left sidebar, look for the **File Explorers** icon (top left).
3. Click the tiny "Page" icon to **Create a New File**.
4. Name the file: LabCoin.sol







Part 2: Token Creation (The Smart Contract)

Copy and paste the following code into your new LabCoin.sol file. This code defines the rules of your cryptocurrency.

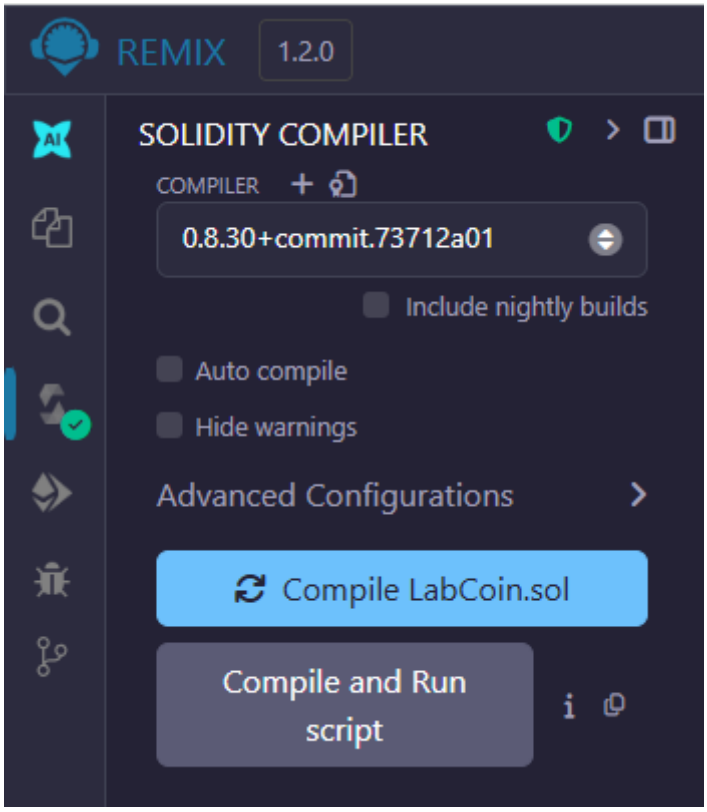
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract LabCoin {
    // 1. TOKEN DETAILS
    string public name = "LabCoin";
    string public symbol = "LAB";
    uint8 public decimals = 18;
    uint256 public totalSupply;

    // 2. THE LEDGER (Who owns what)
    mapping(address => uint256) public balanceOf;

    // Events allow external apps to listen for transfers
    event Transfer(address indexed from, address indexed to, uint256 value);

    // 3. CREATION (Minting)
    // This runs only once when you deploy the contract.
    constructor(uint256 _initialSupply) {
        // Calculate total supply (Supply * 10^18)
        totalSupply = _initialSupply * (10 ** uint256(decimals));

        // Give all tokens to the contract creator (You)
        balanceOf[msg.sender] = totalSupply;
    }

    // 4. MANAGEMENT (Transferring)
    function transfer(address _to, uint256 _value) public returns (bool success) {
        // Check if sender has enough money
        require(balanceOf[msg.sender] >= _value, "Insufficient balance");

        // Deduct from sender, Add to receiver
        balanceOf[msg.sender] -= _value;
        balanceOf[_to] += _value;
```

```
        // Notify the network
        emit Transfer(msg.sender, _to, _value);
        return true;
    }
}
```

Part 3: Compiling

Before the blockchain can read your code, it must be translated into machine code (Bytecode).

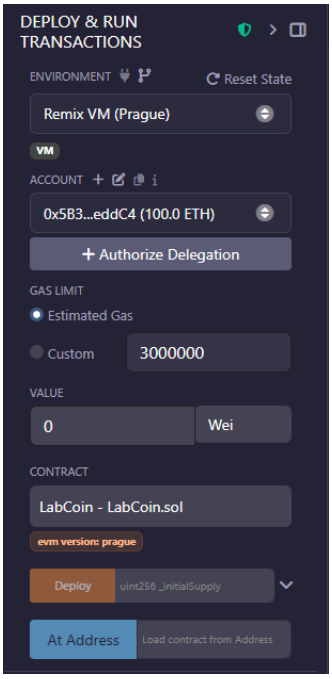
1. Click the **Solidity Compiler** icon on the left sidebar (it looks like an "S" shape).
2. Ensure the "Compiler" version is set to 0.8.x (matching our code).
3. Click the blue button: **Compile LabCoin.sol**.
4. Look for a green checkmark on the sidebar icon. If you see it, your code is valid.



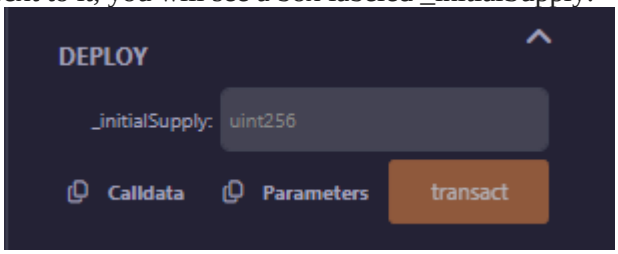
Part 4: Deployment (Creation)

This step simulates launching your token onto the network.

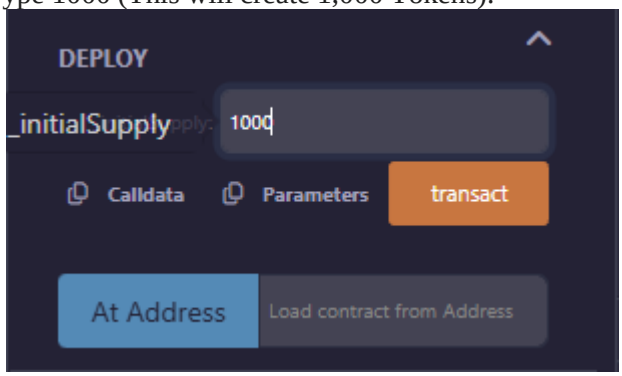
1. Click the **Deploy & Run Transactions** icon on the left sidebar (looks like an Ethereum logo with an arrow).
2. **Environment:** Select Remix VM (Cancun) or Remix VM (London). This is your fake local blockchain.
3. **Account:** You will see a list of accounts (e.g., 0x5B3...). Select the first one. This is "You" (the Admin).



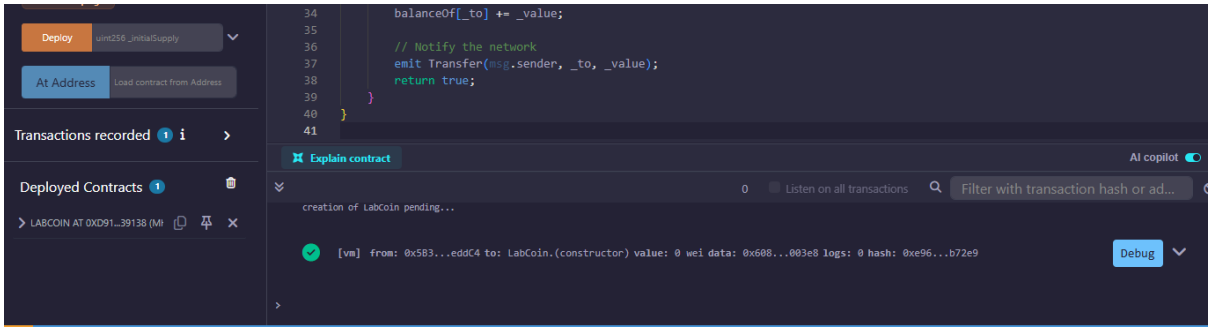
4. **Deploy Section:**
   - Look for the orange **Deploy** button.
   - Next to it, you will see a box labeled _initialSupply.



   - Type 1000 (This will create 1,000 Tokens).



5. Click **Deploy**.

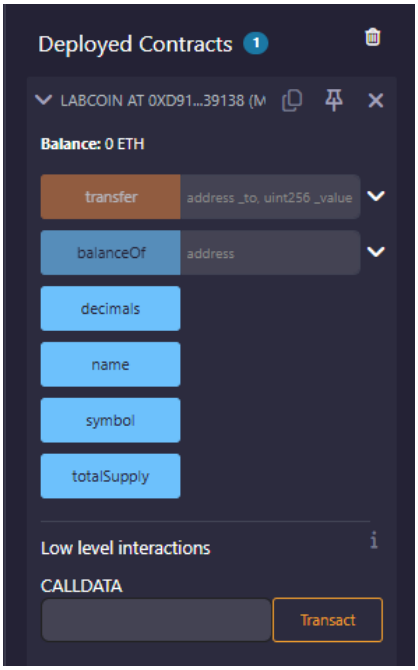


Check the console at the bottom. You should see a green checkmark indicating the transaction succeeded.

Part 5: Token Management

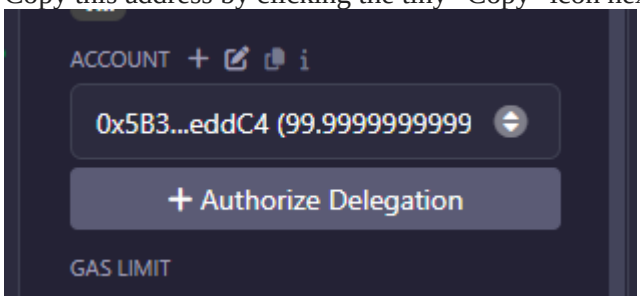
Now that the token exists, let's manage it using the graphical interface Remix generates for us.

1. Scroll down the left sidebar to the **Deployed Contracts** section.
2. Click the arrow > next to LabCoin to expand it. You will see orange and blue buttons.

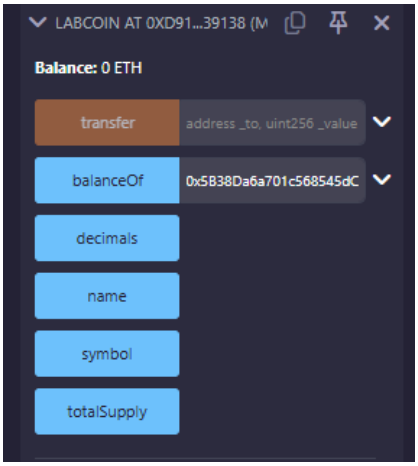


Task A: Check Initial Balance

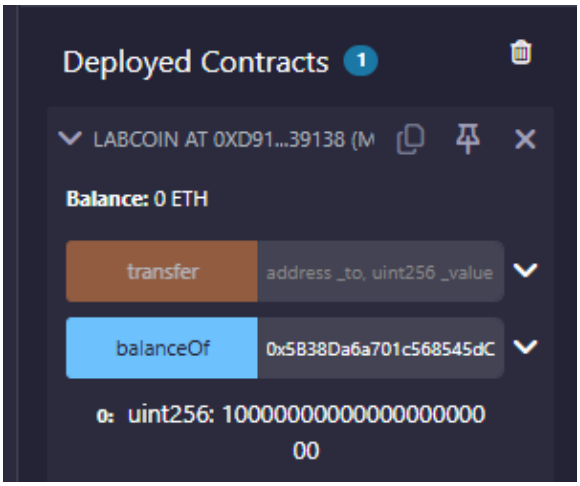
1. In the **Account** dropdown at the top, verify you are still selected on the first account (The Creator).
2. Copy this address by clicking the tiny "Copy" icon next to the account dropdown.



3. Go down to the blue **balanceOf** button.
4. Paste your address into the box and click **balanceOf**.



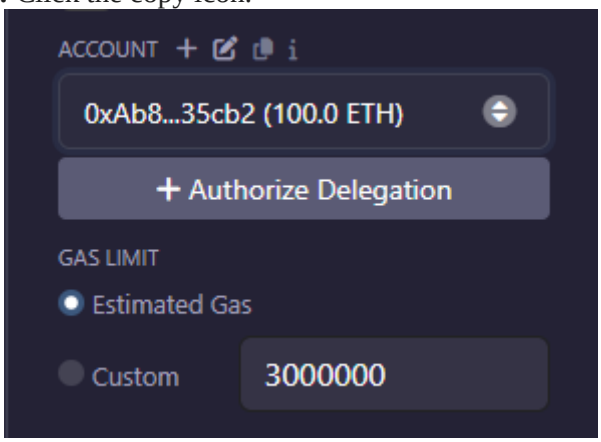
5. **Result:** You should see a huge number.
   - *Why so big?* Because Ethereum doesn't use decimals (like 10.5). It uses integers. 1000 tokens is represented as 1000 followed by 18 zeros.



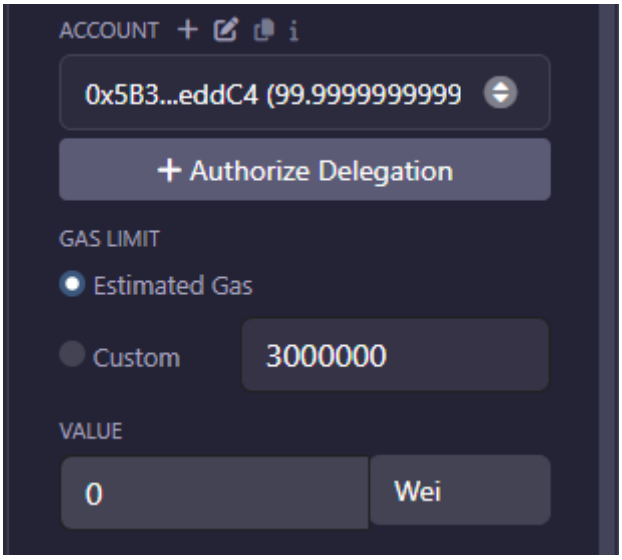
Task B: Transfer Tokens

Let's send tokens to a second user ("Alice").

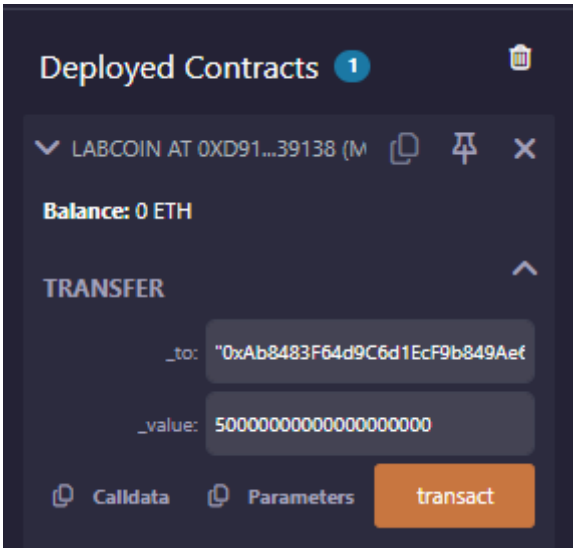
1. **Select Alice:** Go to the **Account** dropdown at the top and select the **second** account in the list.
2. **Copy Alice's Address:** Click the copy icon.

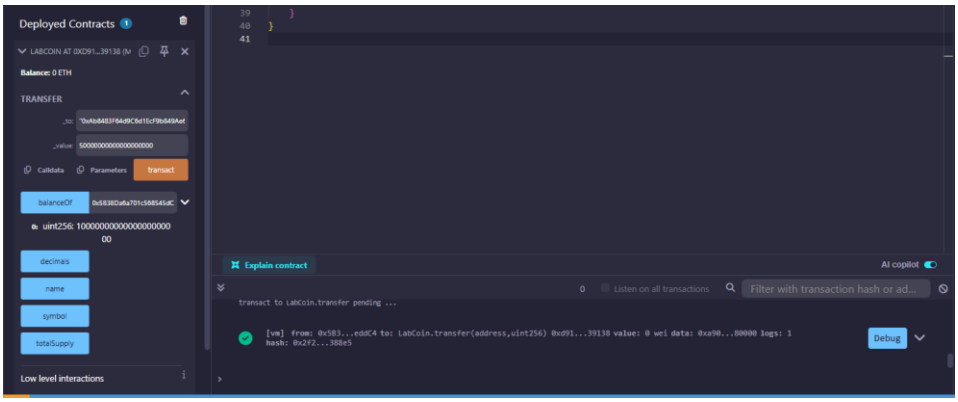


3. **Switch Back:** Select the **first** account again (The Creator) because *you* are the one sending money.



4. **Perform Transfer:**
   - Go to the orange **transfer** button.
   - **_to:** Paste Alice's address.
   - **_value:** We want to send 50 tokens. We must add the zeros manually. Type: 50000000000000000000 (That is 50 followed by 18 zeros).
   - Click **transact**.

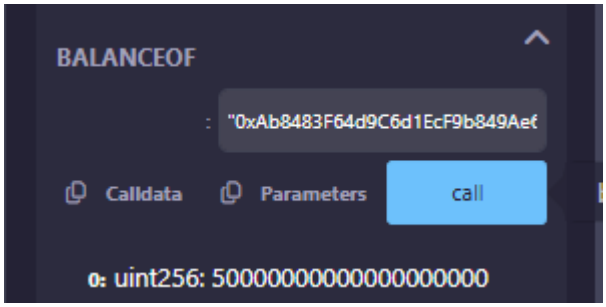




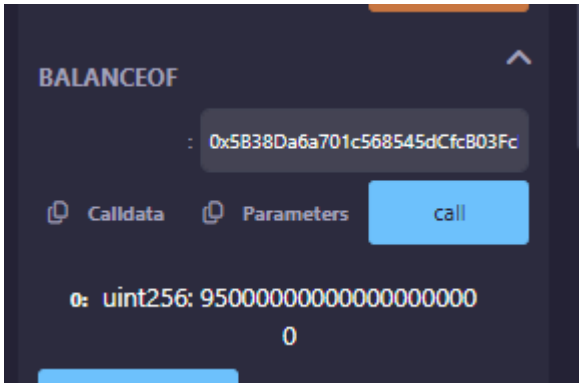
Task C: Validate the Transfer

1. Go back to the blue **balanceOf** button.

2. Paste Alice's address (the second account) and click.
3. **Result:** You should see 50000000000000000000.



4. Now paste your address (the first account) and click.
5. **Result:** Your balance should have decreased.



Conclusion

You have successfully:

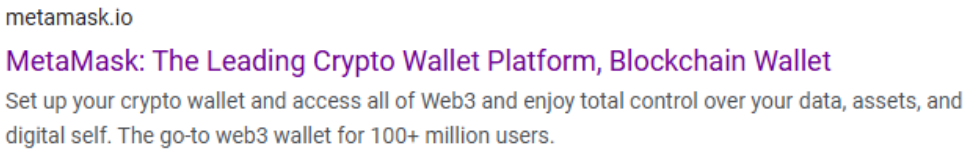
1. **Written** a Smart Contract in Solidity.
2. **Compiled** it into bytecode.
3. **Deployed** (Created) a token on a simulated blockchain.
4. **Managed** the token by executing a transfer transaction.

EXPERIMENT 9

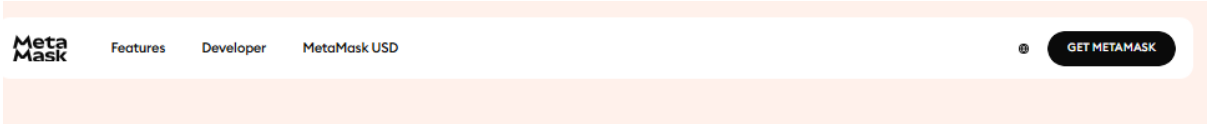
SETUP METAMASK IN THE SYSTEM AND CREATE A WALLET IN THE METAMASK WITH TEST NETWORK

INSTALL METAMASK

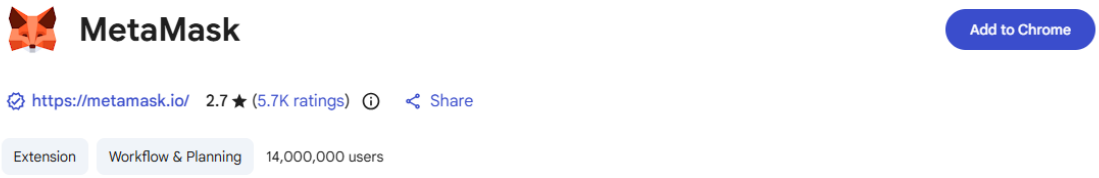
1. Open your preferred browser (Chrome, Firefox, Edge, Brave).



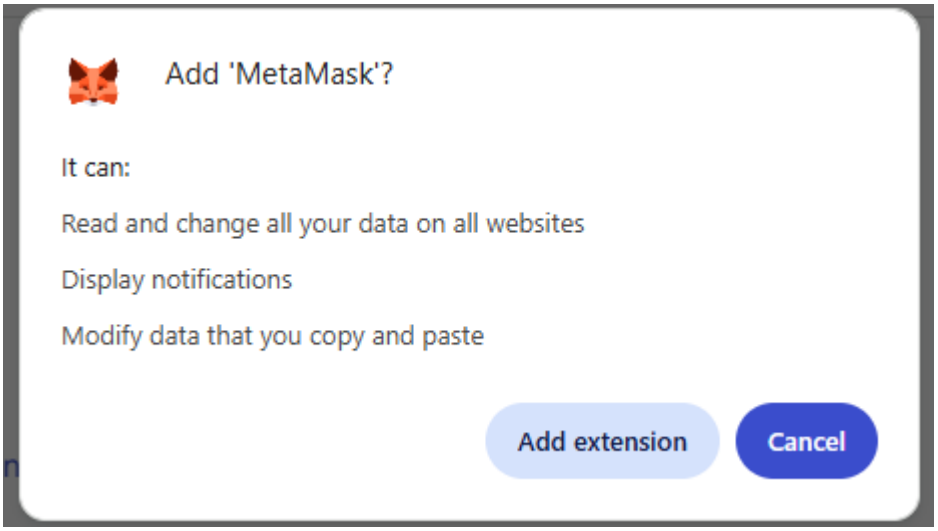
2. Go to the official MetaMask extension page for your browser (search for “MetaMask extension” and ensure the publisher is MetaMask / ConsenSys).



3. Click **Add to Browser** / **Install** and accept the extension permissions.

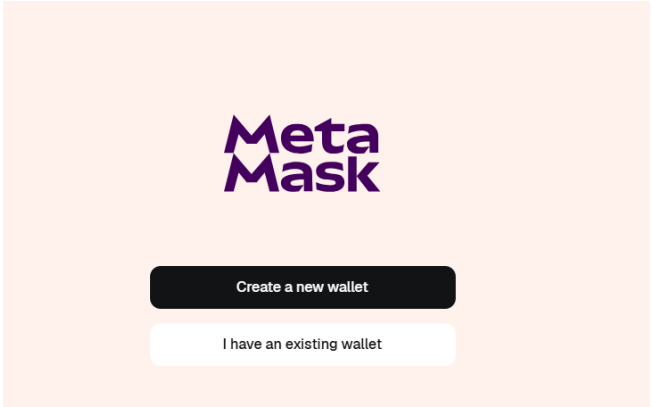


4. After installation, the MetaMask fox icon will appear in your browser toolbar. Click it to open MetaMask

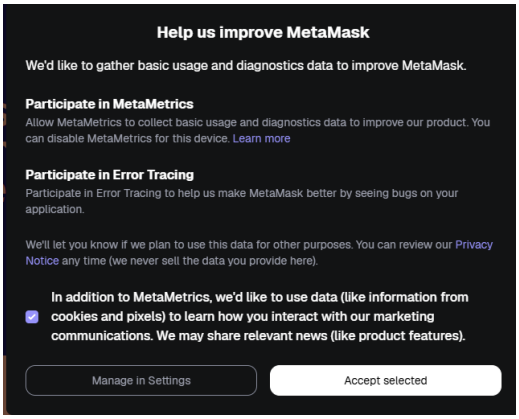


CREATE A NEW WALLET

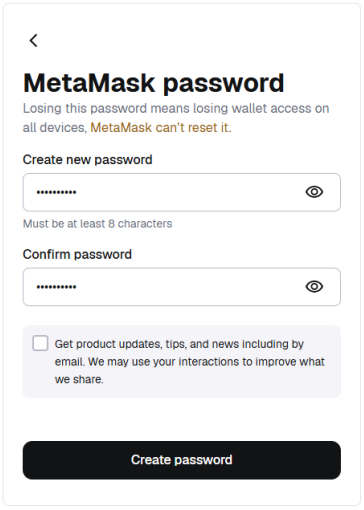
1. Click the MetaMask fox icon and choose **Get Started**.
2. Choose **Create a Wallet**.



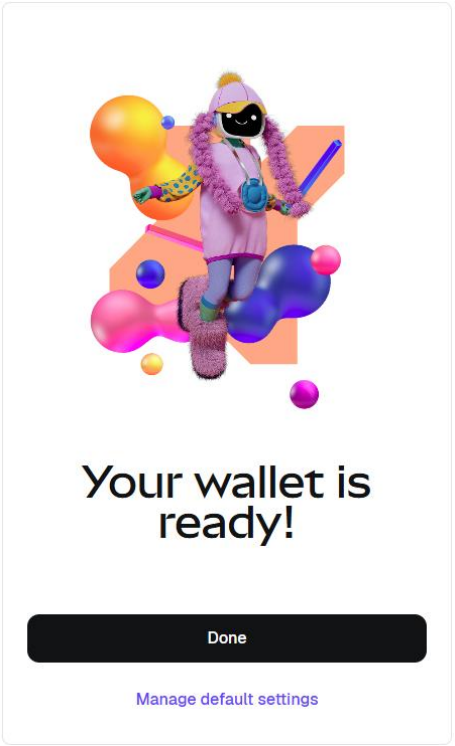
3. MetaMask may ask about data collection — choose your preference.



4. Create a strong password for this browser extension. This password **only unlocks MetaMask on this device**. It is not the seed phrase. Click **Create**.

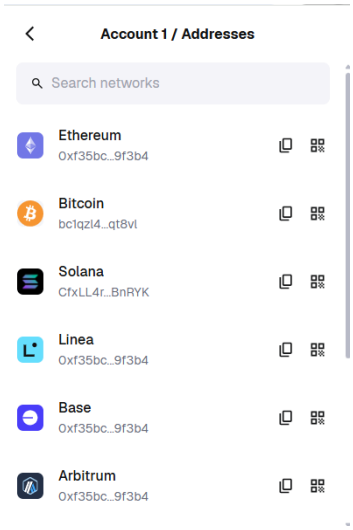


5. MetaMask will show you the Secret Recovery Phrase (seed phrase) in a 12-word (or sometimes 24-word) sequence. **Back it up now**:
    - Write the words on paper (offline) and store in a secure place (safe or locked drawer).
    - Do **not** store the seed phrase in plain text on your computer, email, cloud storage, or take a screenshot.
6. MetaMask will ask you to confirm the recovery phrase by selecting the words in order. Complete this to finish wallet creation.
7. After confirmation, MetaMask shows your new account (Account 1) and its public address (click to copy).

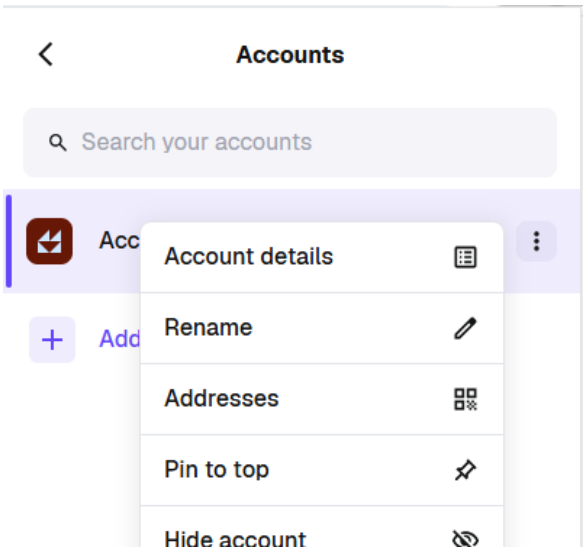


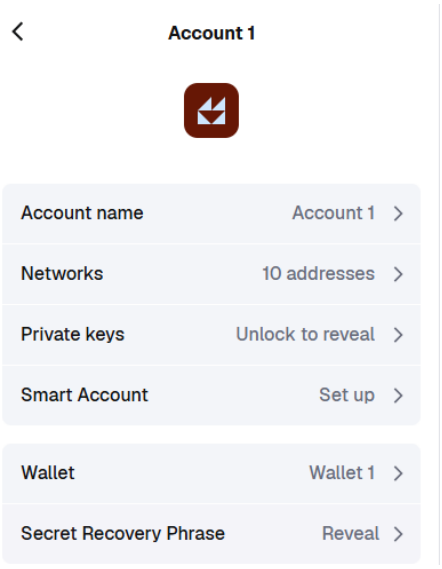
REVEAL & COPY YOUR ACCOUNT ADDRESS / EXPORT PRIVATE KEY

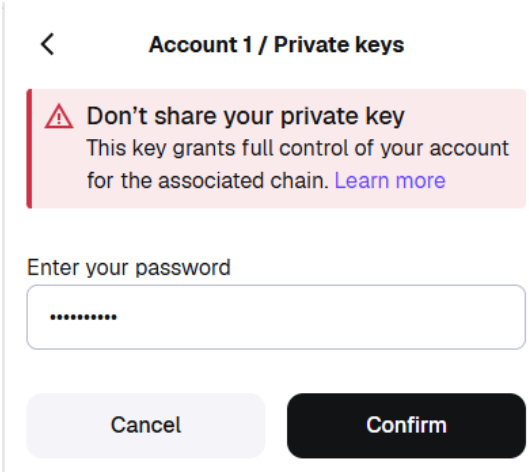
- To see your account address: open MetaMask, click the account name — the address will be copied.



- To export the private key (NOT recommended unless you need it for local dev): Account → Account details → **Export Private Key** → enter your MetaMask password → copy key. **Do not share**. If used in code, keep it in .env locally and never push to git.



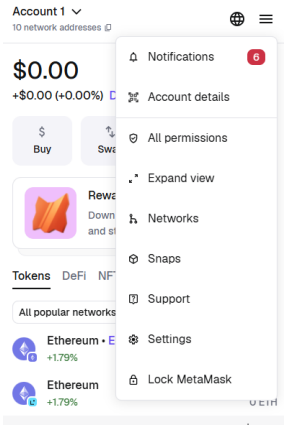




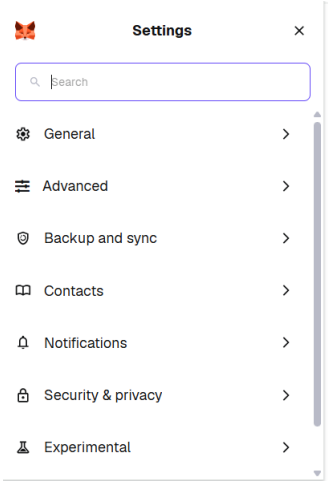
ENABLE TEST NETWORKS IN METAMASK (SHOW SEPOLIA OR OTHER TESTNETS)

Some builds of MetaMask hide test networks by default. To enable them:

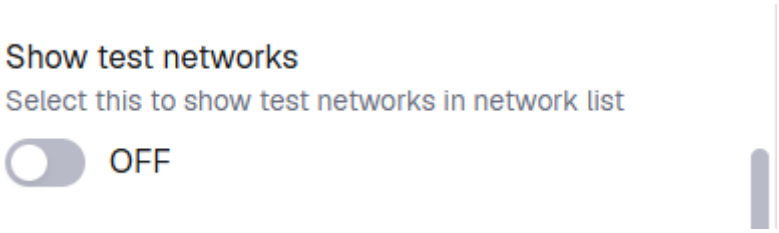
1. Click the account icon (top right of the extension) → **Settings**.



2. Go to **Advanced**.



3. Turn on **Show test networks** (or a similarly-named toggle).

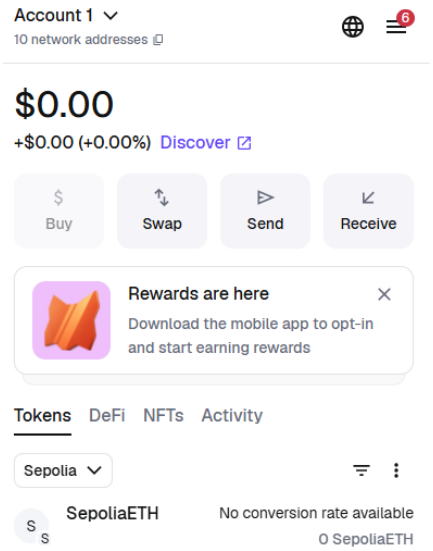


4. Close settings. Now, open the network dropdown at top of the MetaMask popup and you should see testnets like **Sepolia** (and others).

SWITCH METAMASK TO SEPOLIA (TEST NETWORK)

1. Click the network dropdown (top center of MetaMask popup).
2. Select **Sepolia Test Network**.
3. Your account now operates on Sepolia (public testnet). All funds here are test ETH and have no monetary value.

If Sepolia is not listed, add it manually



NOTE: ADD A CUSTOM RPC / ADD SEPOLIA MANUALLY

If you prefer or Sepolia isn't listed:

1. Network dropdown → **Add Network** → **Add a network manually** (or Settings → Networks → Add Network).
2. Fill fields (example for Sepolia):
   - Network name: `Sepolia Testnet`
   - RPC URL: `https://sepolia.infura.io/v3/YOUR_INFURA_KEY` (replace with your provider URL like Alchemy or Infura)

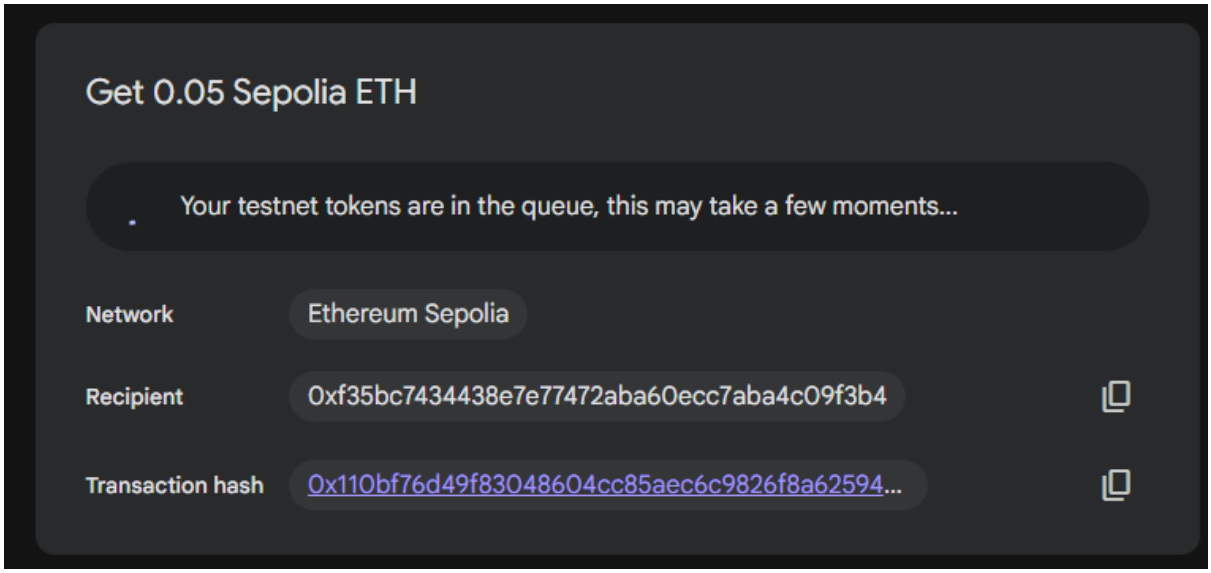
- Chain ID: `11155111`
- Currency symbol: `ETH`
- Block explorer URL (optional): `https://sepolia.etherscan.io`

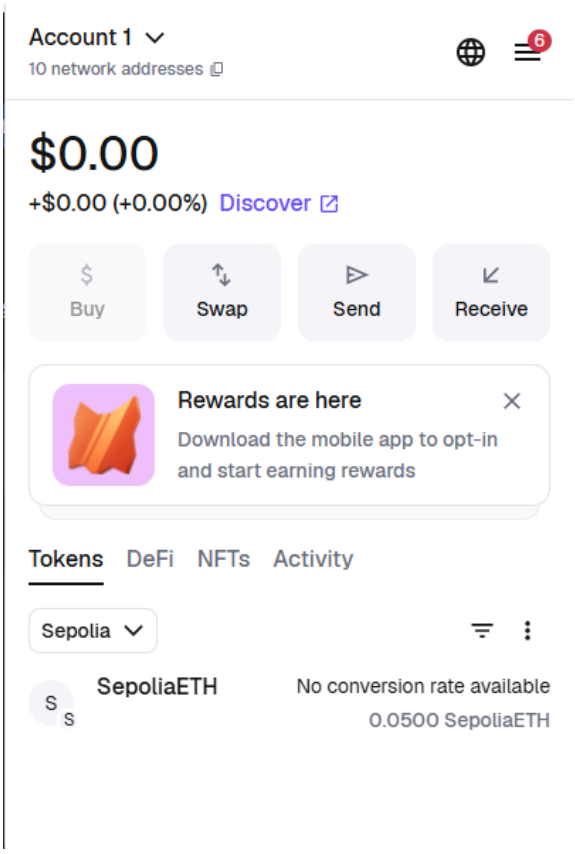
3. Save. Now choose this network in the dropdown.

Replace YOUR_INFURA_KEY / provider URL with your provider's Sepolia RPC endpoint (Alchemy, Infura, or your own node).

GET TEST ETH (SEPOLIA FAUCET)

1. On Sepolia, ETH is free from test faucets. Search "Sepolia faucet" and use a reputable faucet (some require a GitHub or Twitter login).
2. Paste your Sepolia account address (copied from MetaMask) into the faucet form. Request some test ETH.
3. Wait for faucet transaction(s) to confirm — you'll see incoming transactions in MetaMask and balance increase.





CONNECT METAMASK TO A LOCAL HARDHAT (OR GANACHE) NODE (FOR DEVELOPMENT)

1. Start your local node:
   - Hardhat: `npx hardhat node` (runs at `http://127.0.0.1:8545` by default).
   - Ganache: open Ganache GUI or CLI.

Install and Open Ganache

If you haven't installed it:

- Download Ganache GUI from: **https://trufflesuite.com/ganache/**
- Install and open the application.

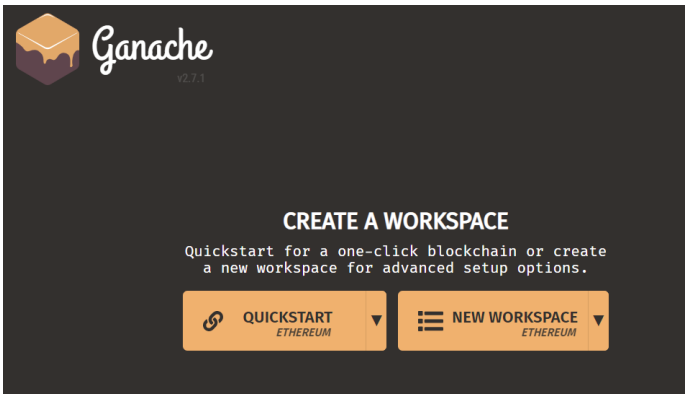
When Ganache opens:

- Click **Quickstart (Ethereum)**
- Ganache will start a local blockchain instantly.

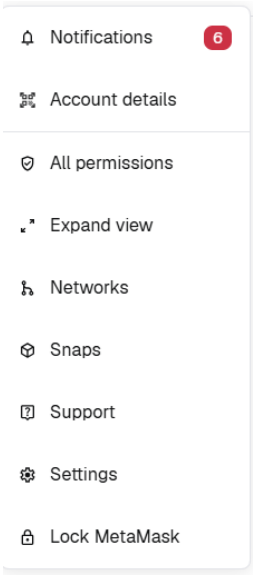
You will now see:

- RPC server address → **http://127.0.0.1:7545**

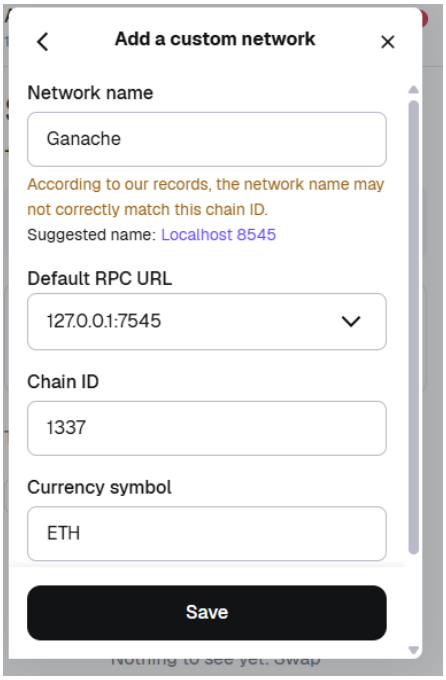
- A list of **10 accounts**, each with **100 ETH** (test ETH)
- Each account has a **private key**



2. In MetaMask: Network dropdown → **Add Network** with:



   - Network name: `Localhost 8545`
   - RPC URL: `http://127.0.0.1:8545`
   - Chain ID: `1337` (or the chain id your dev node prints; Hardhat sometimes uses `31337`)
   - Currency: `ETH`

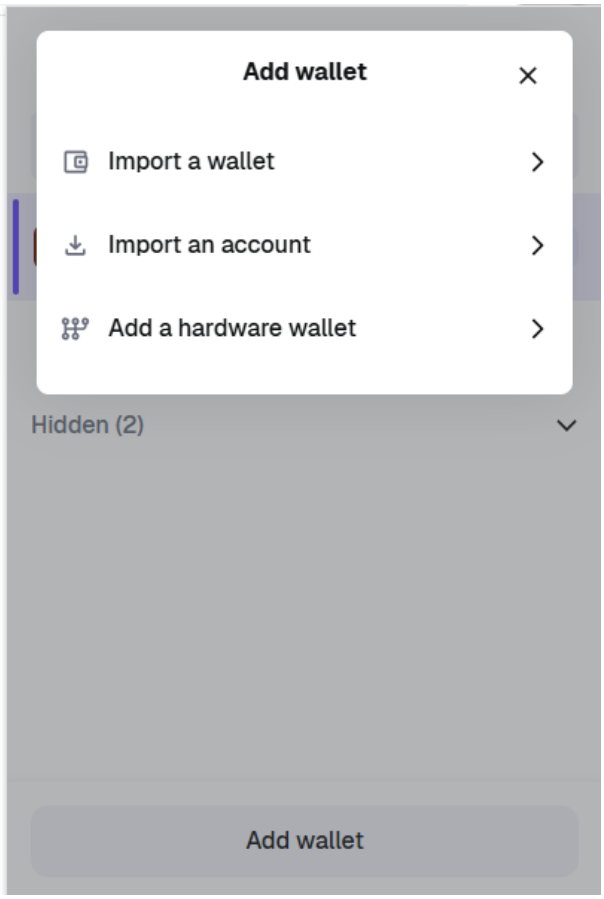


IMPORT A GANACHE ACCOUNT INTO METAMASK

To sign transactions, MetaMask must use one of Ganache's accounts.

1. In Ganache GUI → Click on any account
2. Click **Show Keys**
3. Copy the **Private Key**
   Example:
   `0x4f3edf983ac636a65a842ce7c78d9aa706d3b113b37bb06b1a02d3e268fc8886`

4. Open **MetaMask**
5. Click **Account Icon (top right)**



6. Click **Import Account**
7. Paste the **private key**
8. Click **Import**

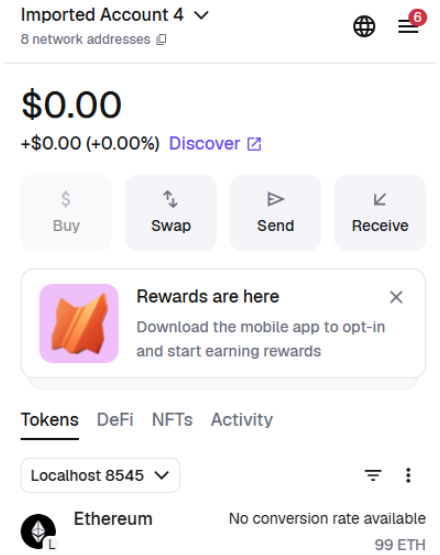
Now MetaMask will show:

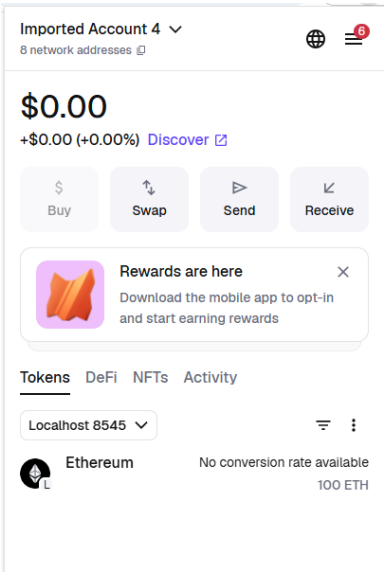
- The Ganache account
- With **100 test ETH** (the balance Ganache assigns)

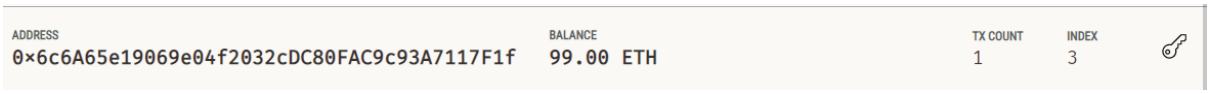
3. Save and switch to this network.
4. Import one of the local node accounts (so MetaMask can sign txs) by using its private key:
    - MetaMask → Account icon → **Import Account** → paste the private key provided by Hardhat or Ganache → **Import**.
    - Now you can send transactions to/from your local dev node through MetaMask.

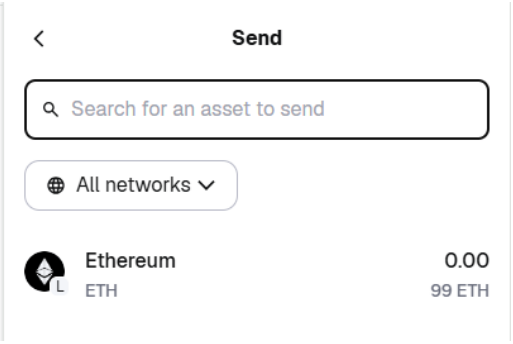
VERIFY CONNECTION & MAKE A TEST TRANSFER

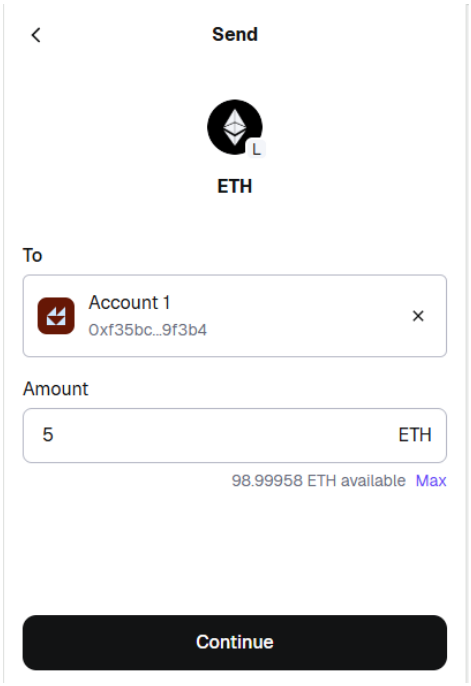
1. With Sepolia selected, click **Activity** / **Assets** — you should see ETH balance and transactions.
2. Send a small amount of test ETH to another test address (or back to yourself) to confirm transactions work.

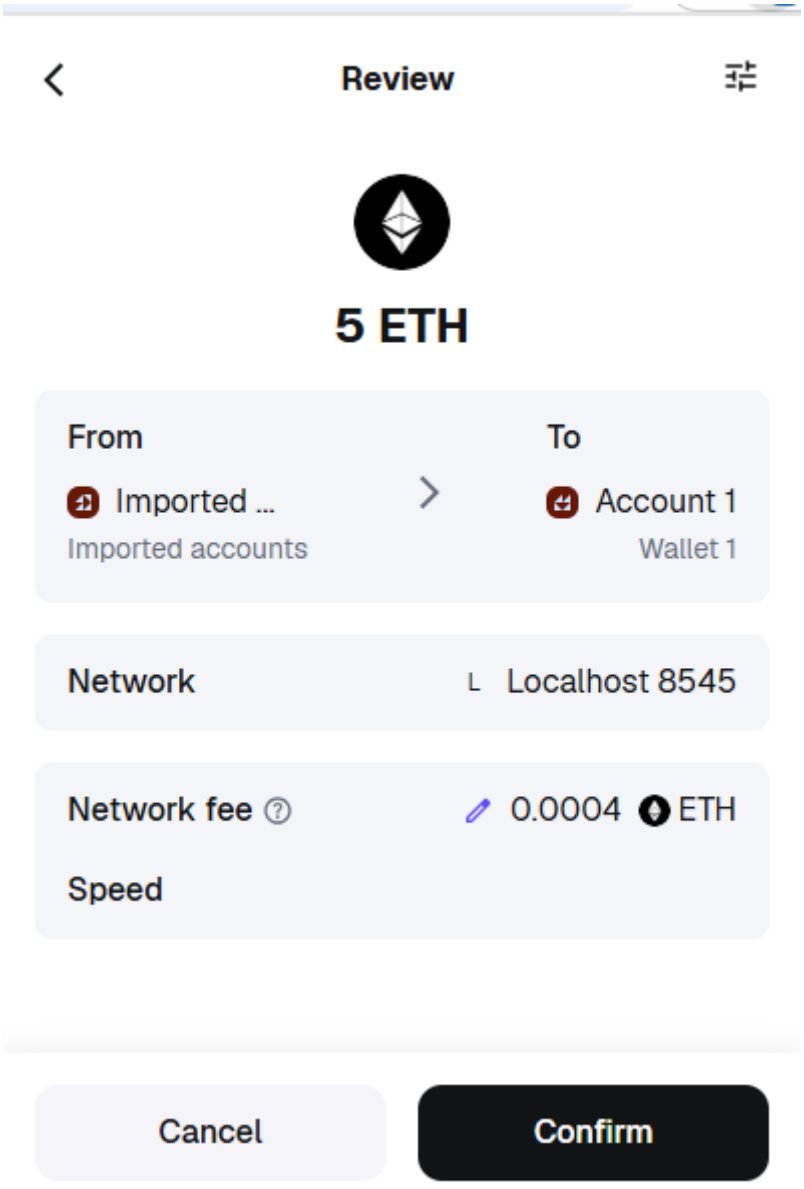

Imported Account 4
8 network addresses
$0.00
+$0.00 (+0.00%) Discover
Buy
Swap
Send
Receive
Rewards are here
Download the mobile app to opt-in and start earning rewards
Tokens
DeFi
NFTs
Activity
Localhost 8545
Ethereum
No conversion rate available
99 ETH


Imported Account 4
8 network addresses
$0.00
+$0.00 (+0.00%) Discover
Buy
Swap
Send
Receive
Rewards are here
Download the mobile app to opt-in and start earning rewards
Tokens
DeFi
NFTs
Activity
Localhost 8545
Ethereum
No conversion rate available
100 ETH


ADDRESS
0×6c6A65e19069e04f2032cDC80FAC9c93A7117F1f
BALANCE
99.00 ETH
TX COUNT
1
INDEX
3


Send
Search for an asset to send
All networks
Ethereum
ETH
0.00
99 ETH


Send
ETH
To
Account 1
0xf35bc...9f3b4
Amount
5
ETH
98.99958 ETH available
Max
Continue


Review
5 ETH
From
Imported ...
Imported accounts
To
Account 1
Wallet 1
Network
Localhost 8545
Network fee
0.0004 ETH
Speed
Cancel
Confirm

Imported Account 4

8 network addresses

$0.00

+$0.00 (+0.00%) Discover

Buy | Swap | Send | Receive

Rewards are here

Download the mobile app to opt-in and start earning rewards

Tokens DeFi NFTs Activity

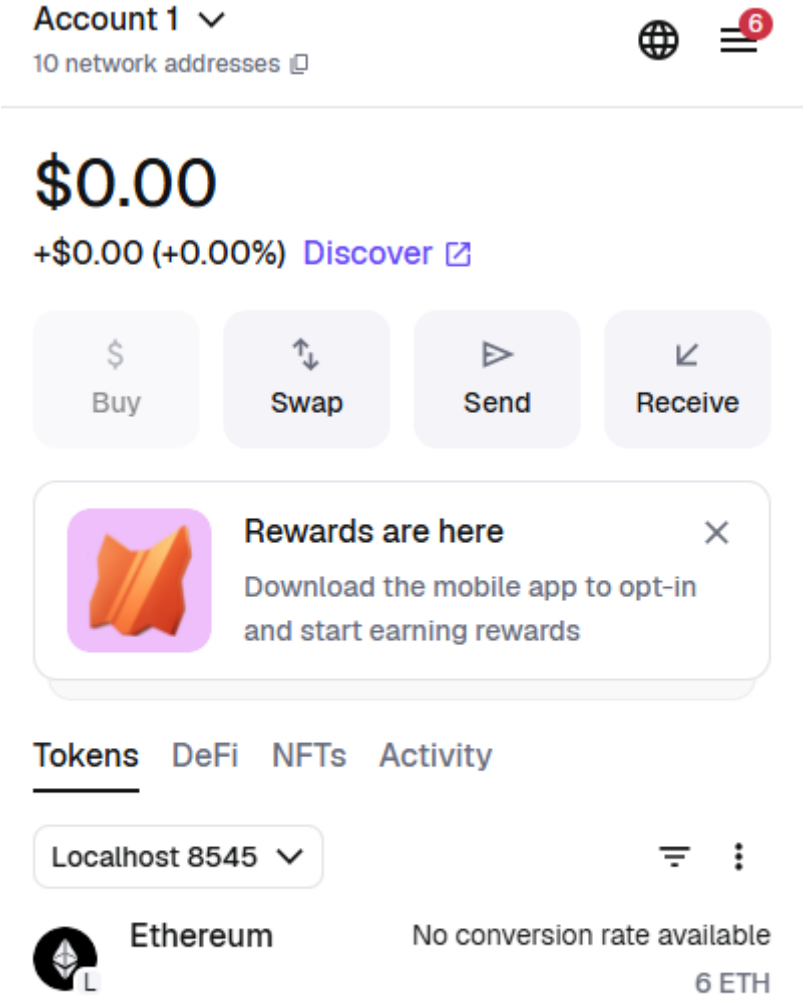
Localhost 8545

Nov 24, 2025

Sent -5 ETH

Confirmed -$14,175.40

Sent -1 ETH



ADDRESS	BALANCE	TX COUNT	INDEX
0×6c6A65e19069e04f2032cDC80FAC9c93A7117F1f	94.00 ETH	2	3

EXPERIMENT 10

DEVELOP A PROGRAM TO IMPLEMENT BLOCKCHAIN IN MERKLE TREES

A Merkle Tree (also called a Hash Tree) is a binary tree data structure used to efficiently summarize and verify the integrity of large sets of data.
It was invented by Ralph Merkle in 1979.

In a Merkle Tree:

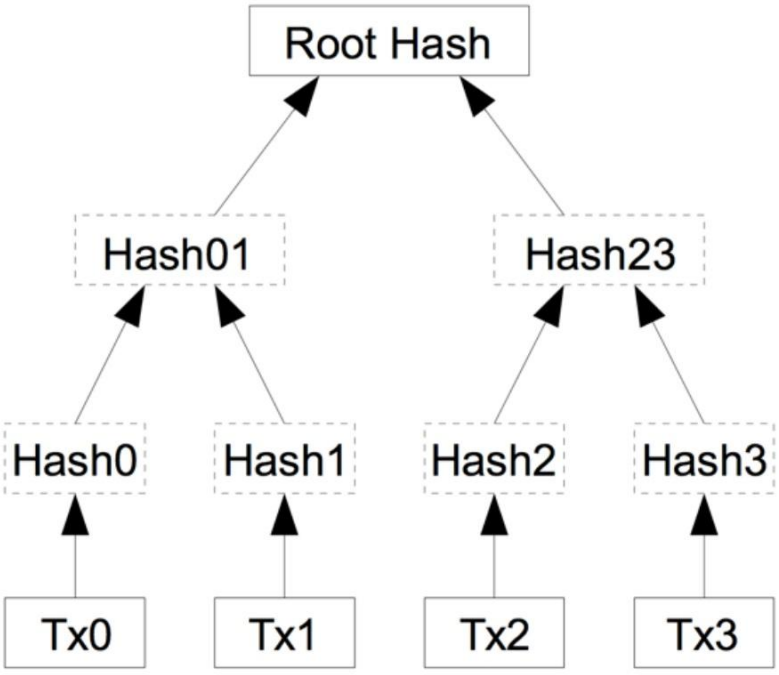
- **Leaves** contain the hashes of the actual data blocks.
- **Parent nodes** contain the hash of the concatenation of their child hashes.
- The **root node** is called the **Merkle Root**, a single hash representing all data in the tree.

Merkle Trees solve fundamental problems in distributed and cryptographic systems:

1. Efficient Data Verification
Only a small number of hashes are required to verify whether a particular transaction/data block is part of the set.

2. Integrity Protection
If any data item changes, its hash changes → parent hash changes → Merkle root changes, detecting tampering.

3. Scalability
Ideal for systems with very large datasets (e.g., millions of transactions in blockchain).



```
# merkle_blockchain_short.py — compact version

import hashlib, json, time
from typing import List, Tuple

def sha256(b: bytes) -> str: return hashlib.sha256(b).hexdigest()
def hstr(s: str) -> str: return sha256(s.encode())

class MerkleTree:
    def __init__(self, leaves: List[str]):
        self.leaves = [hstr(l) for l in leaves]
        self.levels = [self.leaves] if self.leaves else []
        self.build()

    def build(self):
        cur = self.leaves.copy()
        while len(cur) > 1:
            nxt = []
            for i in range(0, len(cur), 2):
                a = cur[i]; b = cur[i+1] if i+1 < len(cur) else a
                nxt.append(sha256(bytes.fromhex(a) + bytes.fromhex(b)))
            cur = nxt
            self.levels.append(cur)

    def root(self) -> str: return self.levels[-1][0] if self.levels else ''
    def proof(self, idx: int) -> List[Tuple[str,str]]:
        if not (0 <= idx < len(self.leaves)): raise IndexError
```

```
        p=[]; i=idx
        for lvl in self.levels[:-1]:
            sib = i+1 if i%2==0 else i-1
            if sib >= len(lvl): sib = i
            p.append((lvl[sib], 'right' if i%2==0 else 'left'))
            i//=2
        return p

    @staticmethod
    def verify(leaf: str, proof: List[Tuple[str,str]], root: str) -> bool:
        cur = hstr(leaf)
        for s, pos in proof:
            cur = sha256(bytes.fromhex(cur) + bytes.fromhex(s)) if pos=='right' else
sha256(bytes.fromhex(s) + bytes.fromhex(cur))
        return cur == root


class Block:
    def __init__(self, idx:int, txs:List[str], prev:str, diff:int=2):
        self.idx, self.txs, self.prev, self.diff = idx, txs[:], prev, diff
        self.ts = time.time(); self.nonce = 0
        self.merkle = MerkleTree(self.txs).root()
        self.hash = self._hash()

    def _hash(self) -> str:
        obj =
{'idx':self.idx,'ts':self.ts,'prev':self.prev,'nonce':self.nonce,'merkle':self.merkle,'cnt':len(self.txs
)}
        return sha256(json.dumps(obj, sort_keys=True).encode())
```

```
    def mine(self):
        target = '0'*self.diff
        while True:
            self.hash = self._hash()
            if self.hash.startswith(target): break
            self.nonce += 1


class Blockchain:
    def __init__(self, diff=2):
        self.diff = diff; self.chain: List[Block] = []
        g = Block(0, ['genesis'], '0'*64, diff); g.mine(); self.chain.append(g)


    def add(self, txs:List[str]) -> Block:
        b = Block(len(self.chain), txs, self.chain[-1].hash, self.diff); b.mine();
self.chain.append(b); return b


    def valid(self) -> bool:
        for i in range(1,len(self.chain)):
            b, p = self.chain[i], self.chain[i-1]
            if b.prev != p.hash: return False
            if not b.hash.startswith('0'*b.diff): return False
            if MerkleTree(b.txs).root() != b.merkle: return False
            if b._hash() != b.hash: return False
        return True


if __name__ == '__main__':
```

```
bc = Blockchain(diff=3)
b1 = bc.add(["Alice->Bob:5","Carol->Dave:2","Eve->Frank:7"])
print("Block", b1.idx, "hash", b1.hash, "merkle", b1.merkle)
b2 = bc.add(["Ivy->John:3","Alice->Carol:1"])
print("Block", b2.idx, "hash", b2.hash)
print("Chain valid?", bc.valid())


# Merkle proof example
proof = MerkleTree(b1.txs).proof(1)
print("Proof for tx:", b1.txs[1])
print(proof)
print("Verify:", MerkleTree.verify(b1.txs[1], proof, b1.merkle))
```

OUTPUT

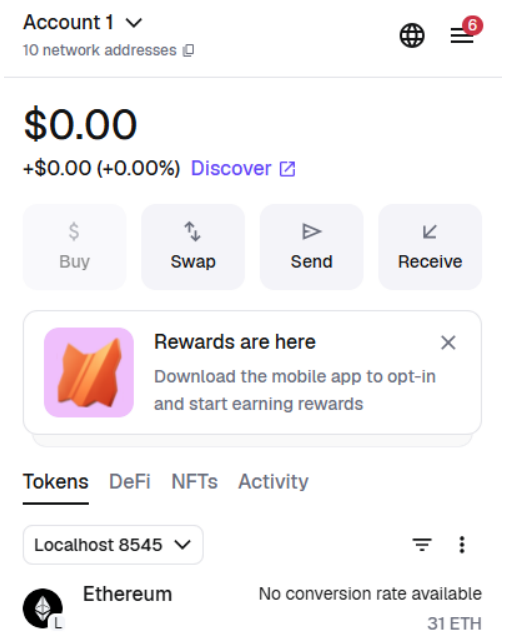
```
Added block 1 with hash: 000c6384a4745b0b7dbf3efc1626f9ce54154f5bd2615b491f4a85fd7acf5e8e
Merkle root: 758831ab1038918f01fe2fe6aefd78e429657bd684220e6317c6bbe9319eb45e

Added block 2 with hash: 0000ac6d22cdaee89f155727553f93007c15caf9dcb31c17265ab68f94e1c293
Merkle root: d889bf28f19b9af73e4f750f5c120ffd6557e81f023d9add949cdf0f2349aa5c

Blockchain valid? True
Transaction to prove: 'Charlie pays Dave 2'
Proof (sibling_hash, position):
  66ed59ebf7d156f0f5fca203039b6263ebbc7da0f2479caaa04544864d7b1719 left
  25dc953f719a831fa86d2d489a64b8db9dfede052b3db0a47d8e00b922f913c1 right
Merkle proof verification result: True
```

EXPERIMENT 11

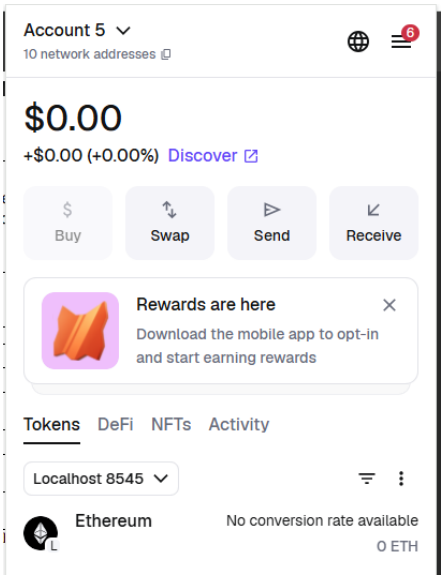
CREATE MULTIPLE ACCOUNTS IN METAMASK AND PERFORM THE BALANCE TRANSFER BETWEEN THE ACCOUNTS AND DESCRIBE THE TRANSACTION SPECIFICATION

Follow the same steps as in experiment 9, but the difference is to transfer between 2 accounts. Initially transfer some ETHS from imported account from Ganache as in case of experiment 9.Then transfer the balance between the 2 accounts in Metamask.

Account 1 has 31 ETH initially which was transferred from Ganache account.

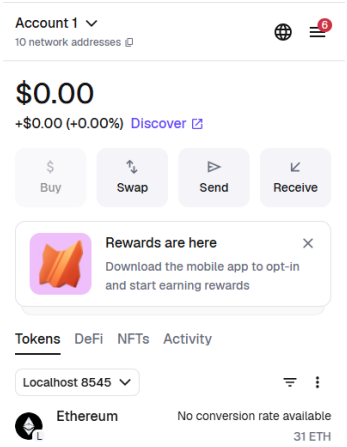


Create a new account which has 0 ETH.

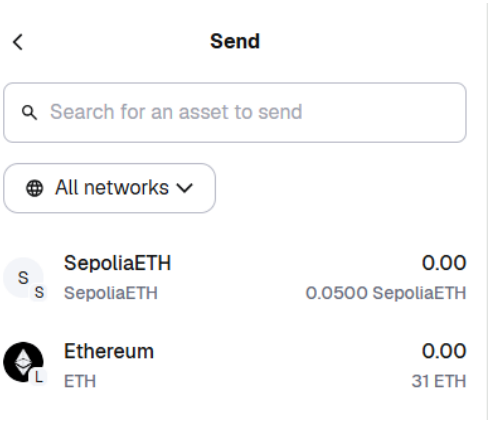


Now transfer some ETHs from Account1 to Account5

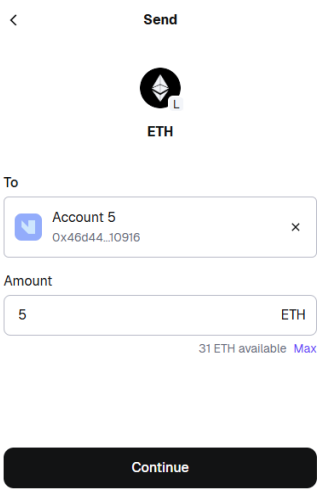
Click on Account1 and then select send



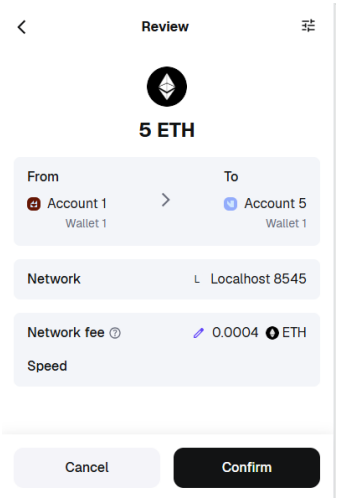
Select Ethereum



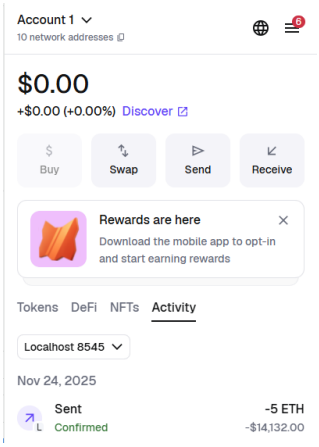
Copy Account 5 address and paste it and enter the number of ETH’s you want to send.



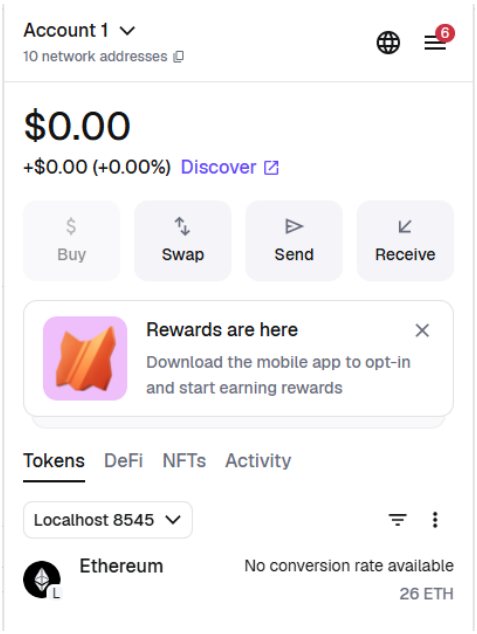
Click confirms to initiate the transaction.

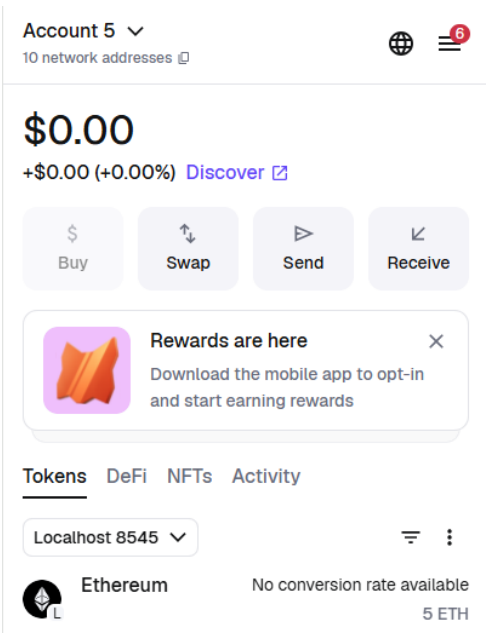


The transfer will be confirmed.



Verify the transfer in Account1 and Account5


Account 1
10 network addresses
6
$0.00
+$0.00 (+0.00%) Discover
Buy
Swap
Send
Receive
Rewards are here
Download the mobile app to opt-in and start earning rewards
Tokens
DeFi
NFTs
Activity
Localhost 8545
Ethereum
No conversion rate available
26 ETH


Account 5
10 network addresses
6
$0.00
+$0.00 (+0.00%) Discover
Buy
Swap
Send
Receive
Rewards are here
Download the mobile app to opt-in and start earning rewards
Tokens
DeFi
NFTs
Activity
Localhost 8545
Ethereum
No conversion rate available
5 ETH