ERC20 Token and Vendor smart contract implementation:

ERC20:

```
pragma solidity 0.8.20; //Do not change the solidity version as it negatively impacts submission grading
// SPDX-License-Identifier: MIT

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

// learn more: https://docs.openzeppelin.com/contracts/4.x/erc20

contract YourToken is ERC20 {
  constructor() ERC20("Gold", "GLD") {
    _mint(msg.sender, 2000 * 10 ** decimals());
  }
}
```

Vendor smart contract:

```
pragma solidity 0.8.20; //Do not change the solidity version as it negatively impacts submission grading
// SPDX-License-Identifier: MIT

import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract Vendor is Ownable {
  event BuyTokens(address buyer, uint256 amountOfETH, uint256 amountOfTokens);
  event SellTokens(address seller, uint256 amountOfTokens, uint256 amountOfETH);

  IERC20 public yourToken;
  uint256 public constant tokensPerEth = 100;

  constructor(address tokenAddress) Ownable(msg.sender) {
    yourToken = IERC20(tokenAddress);
  }

  receive() external payable {}

  function buyTokens() external payable {
    require(msg.value > 0, "Must send ETH to buy tokens");
    uint256 amountOfTokens = msg.value * tokensPerEth;
    require(yourToken.balanceOf(address(this)) >= amountOfTokens, "Vendor has insufficient tokens");
    yourToken.transfer(msg.sender, amountOfTokens);
    emit BuyTokens(msg.sender, msg.value, amountOfTokens);
  }

  function withdraw() external onlyOwner {
    uint256 balance = address(this).balance;
    (bool success, ) = payable(owner()).call{value: balance}("");
    require(success, "Withdrawal failed");
  }

  function sellTokens(uint256 _amount) external {
    require(_amount > 0, "Amount must be greater than 0");
    uint256 ethValue = _amount / tokensPerEth;
```

```
        require(address(this).balance >= ethValue, "Insufficient ETH balance in vendor");
        yourToken.transferFrom(msg.sender, address(this), _amount);
        payable(msg.sender).transfer(ethValue);
        emit SellTokens(msg.sender, _amount, ethValue);
    }
}
```

Deploying smart contracts:

Deploying ERC20 token script:

```
import { HardhatRuntimeEnvironment } from "hardhat/types";
import { DeployFunction } from "hardhat-deploy/types";
// import { Contract } from "ethers";

/**
 * Deploys a contract named "YourToken" using the deployer account and
 * constructor arguments set to the deployer address
 *
 * @param hre HardhatRuntimeEnvironment object.
 */
const deployYourToken: DeployFunction = async function (hre: HardhatRuntimeEnvironment) {
  /*
    On localhost, the deployer account is the one that comes with Hardhat, which is already funded.

    When deploying to live networks (e.g `yarn deploy --network sepolia`), the deployer account
    should have sufficient balance to pay for the gas fees for contract creation.

    You can generate a random account with `yarn generate` which will fill DEPLOYER_PRIVATE_KEY
    with a random private key in the .env file (then used on hardhat.config.ts)
    You can run the `yarn account` command to check your balance in every network.
  */
  const { deployer } = await hre.getNamedAccounts();
  const { deploy } = hre.deployments;

  await deploy("YourToken", {
    from: deployer,
    // Contract constructor arguments
    args: [],
    log: true,
    // autoMine: can be passed to the deploy function to make the deployment process faster on local networks by
    // automatically mining the contract deployment transaction. There is no effect on live networks.
    autoMine: true,
  });

  // Get the deployed contract
  // const yourToken = await hre.ethers.getContract<Contract>("YourToken", deployer);
};

export default deployYourToken;

// Tags are useful if you have multiple deploy files and only want to run one of them.
// e.g. yarn deploy --tags YourToken
deployYourToken.tags = ["YourToken"];
```

Deploying Vendor smart contract script:

```ts
import { HardhatRuntimeEnvironment } from "hardhat/types";
import { DeployFunction } from "hardhat-deploy/types";
import { Contract } from "ethers";

/**
 * Deploys a contract named "Vendor" using the deployer account and
 * constructor arguments set to the deployer address
 *
 * @param hre HardhatRuntimeEnvironment object.
 */
// eslint-disable-next-line @typescript-eslint/no-unused-vars
const deployVendor: DeployFunction = async function (hre: HardhatRuntimeEnvironment) {
  /*
    On localhost, the deployer account is the one that comes with Hardhat, which is already funded.

    When deploying to live networks (e.g `yarn deploy --network goerli`), the deployer account
    should have sufficient balance to pay for the gas fees for contract creation.

    You can generate a random account with `yarn generate` which will fill DEPLOYER_PRIVATE_KEY
    with a random private key in the .env file (then used on hardhat.config.ts)
    You can run the `yarn account` command to check your balance in every network.
  */
  // Deploy Vendor
  const { deployer } = await hre.getNamedAccounts();
  const { deploy } = hre.deployments;
  const yourToken = await hre.ethers.getContract<Contract>("YourToken", deployer);
  const yourTokenAddress = await yourToken.getAddress();
  await deploy("Vendor", {
    from: deployer,
    // Contract constructor arguments
    args: [yourTokenAddress],
    log: true,
    // autoMine: can be passed to the deploy function to make the deployment process faster on local networks by
    // automatically mining the contract deployment transaction. There is no effect on live networks.
    autoMine: true,
  });
  const vendor = await hre.ethers.getContract<Contract>("Vendor", deployer);
  const vendorAddress = await vendor.getAddress();
  // Transfer tokens to Vendor
  await yourToken.transfer(vendorAddress, hre.ethers.parseEther("1000"));
  // Transfer contract ownership to your frontend address
  // Replace with your frontend address
  await vendor.transferOwnership("0x70997970C51812dc3A010C7d01b50e0d17dc79C8");
};

export default deployVendor;

// Tags are useful if you have multiple deploy files and only want to run one of them.
// e.g. yarn deploy --tags Vendor
deployVendor.tags = ["Vendor"];
```

```
E:\school stuff\2025\cong_nghe_BlockChain\practice\hardhat-vendor\challenge-2-token-vendor>yarn deploy
Nothing to compile
No need to generate any newer typings.
deploying "YourToken" (tx: 0x804f9e2ff64f968d120ab069e0c5ad6ff88b375aa657353e7c3e614cef2aff55)...: deployed at 0x5FbDB2315678afecb367f032d93F642f64180aa3 with 557977 gas
deploying "Vendor" (tx: 0x9ff5711b2a84b7988b6015cbc55723a5e3da2d9f9a305e2a8c71f893cad3b2f5)...: deployed at 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512 with 520328 gas
Updated TypeScript contract definition file on ../nextjs/contracts/deployedContracts.ts

E:\school stuff\2025\cong_nghe_BlockChain\practice\hardhat-vendor\challenge-2-token-vendor>
```

Web UI

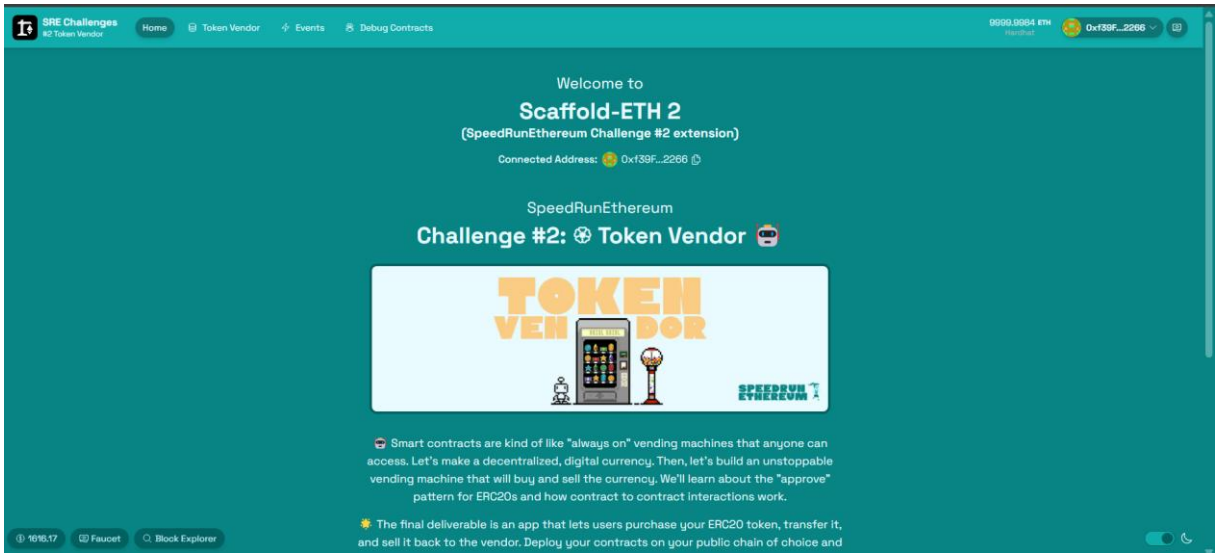
Implemented https://github.com/scaffold-eth/se-2-challenges/tree/challenge-2-token-vendor ‘s Web UI

Starting the web page:

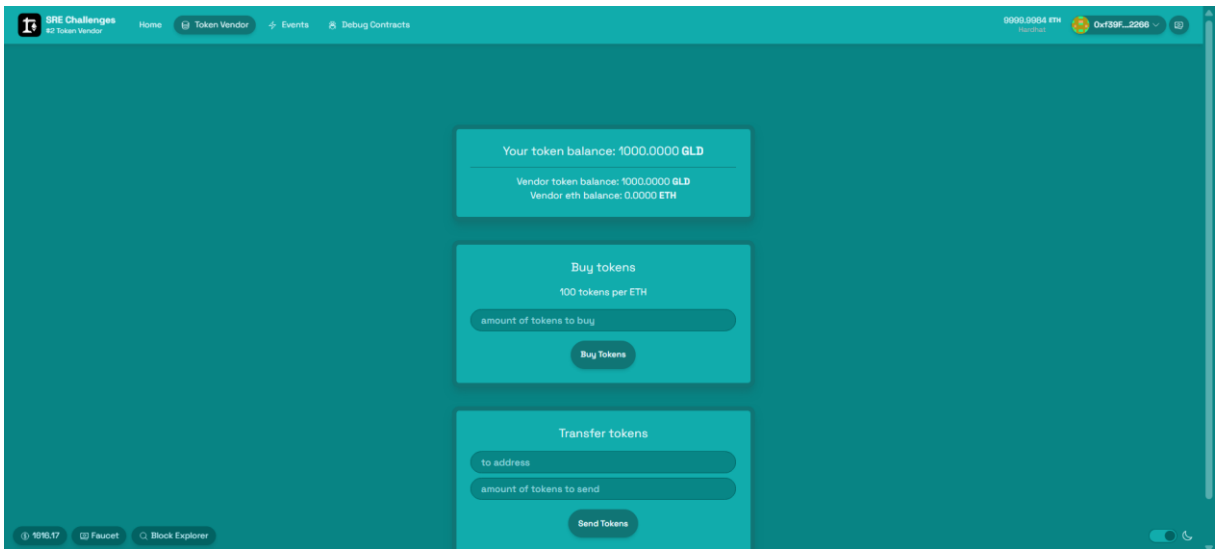
```
E:\school stuff\2025\cong_nghe_BlockChain\practice\hardhat-vendor\challenge-2-token-vendor>yarn start
  ▲ Next.js 14.2.28
  - Local:        http://localhost:3000

 ✓ Starting...
 ✓ Ready in 12.7s
 o Compiling / ...
 ✓ Compiled / in 9.8s (7754 modules)
GET / 200 in 14928ms
HEAD / 200 in 36ms
```

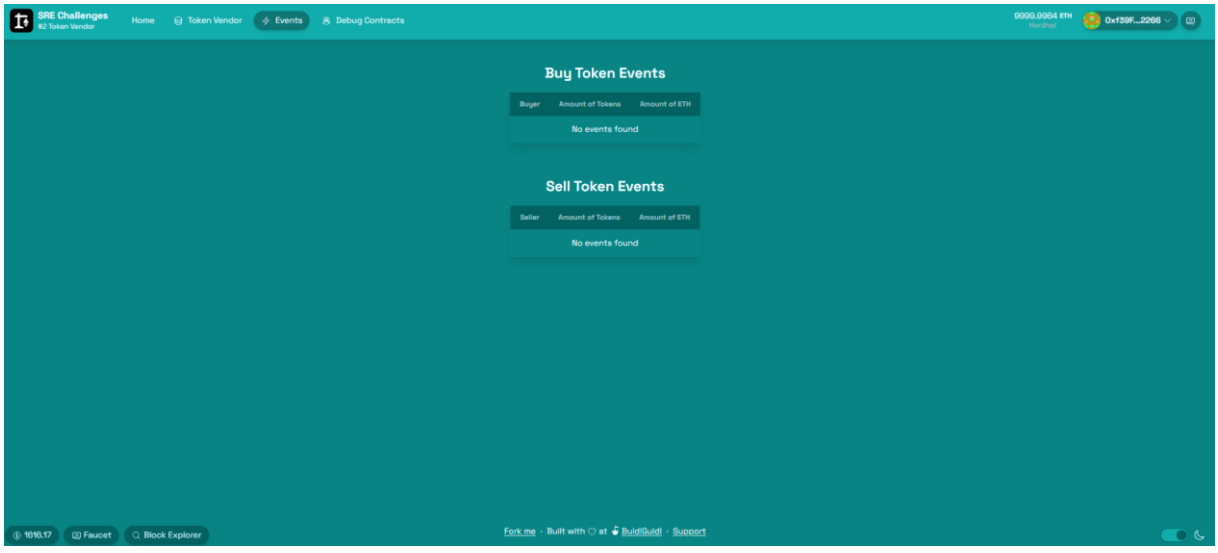
Home page:



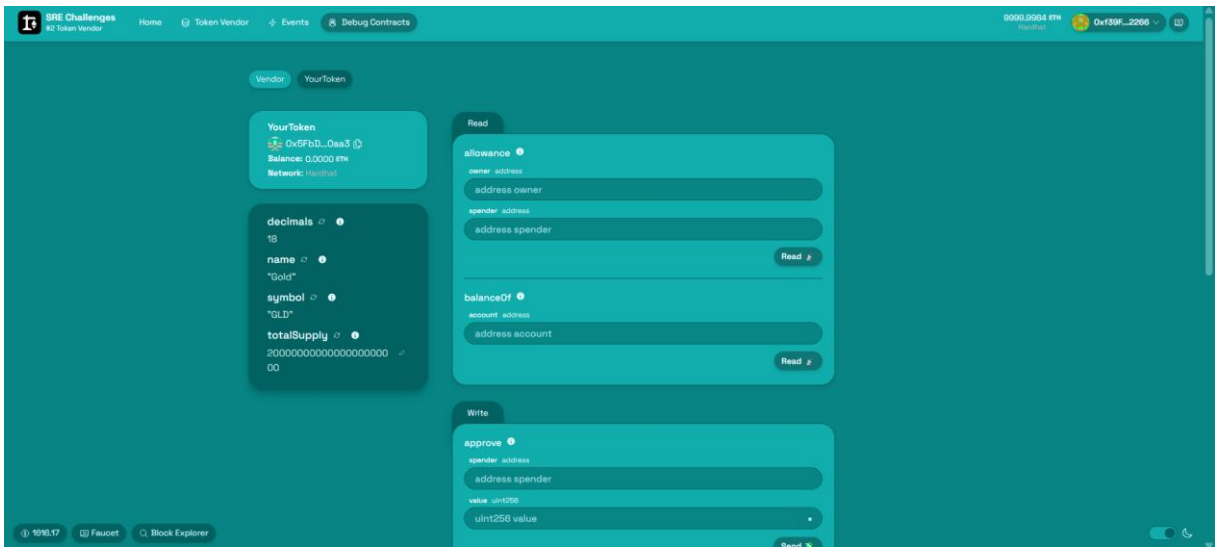
Vendor page:



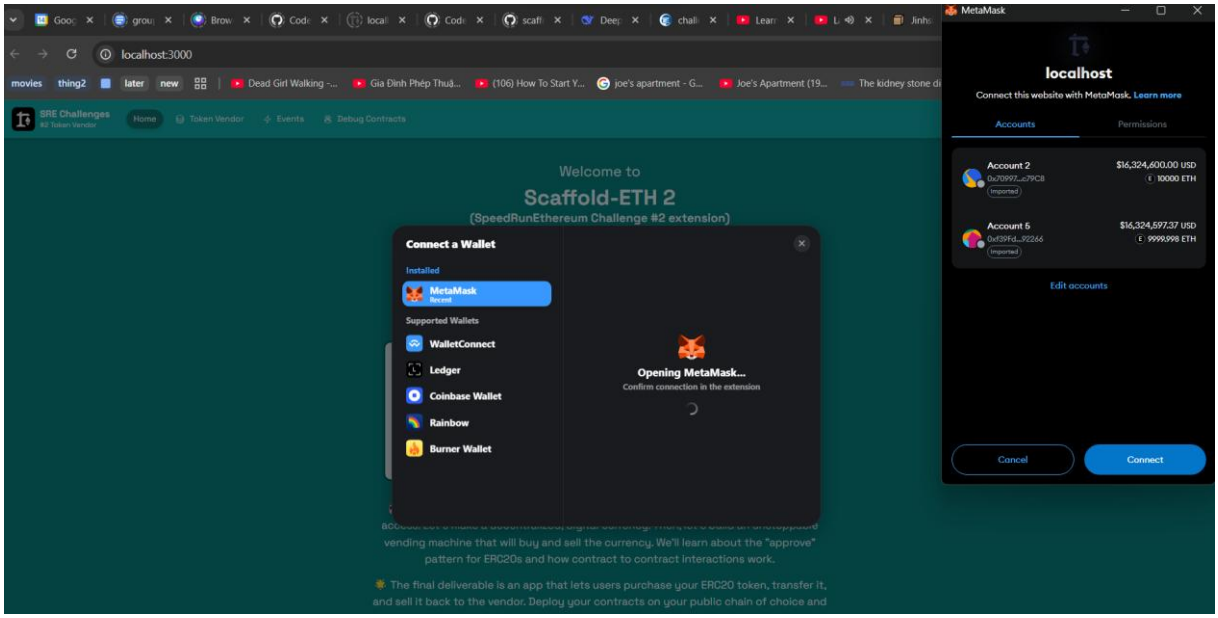
Events page:



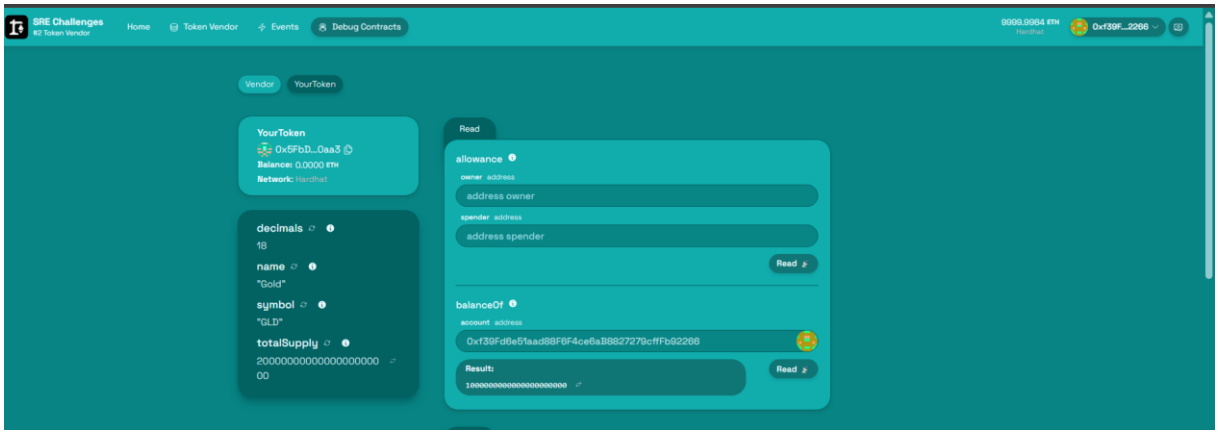
ERC20 Token page:



Login using metamask:

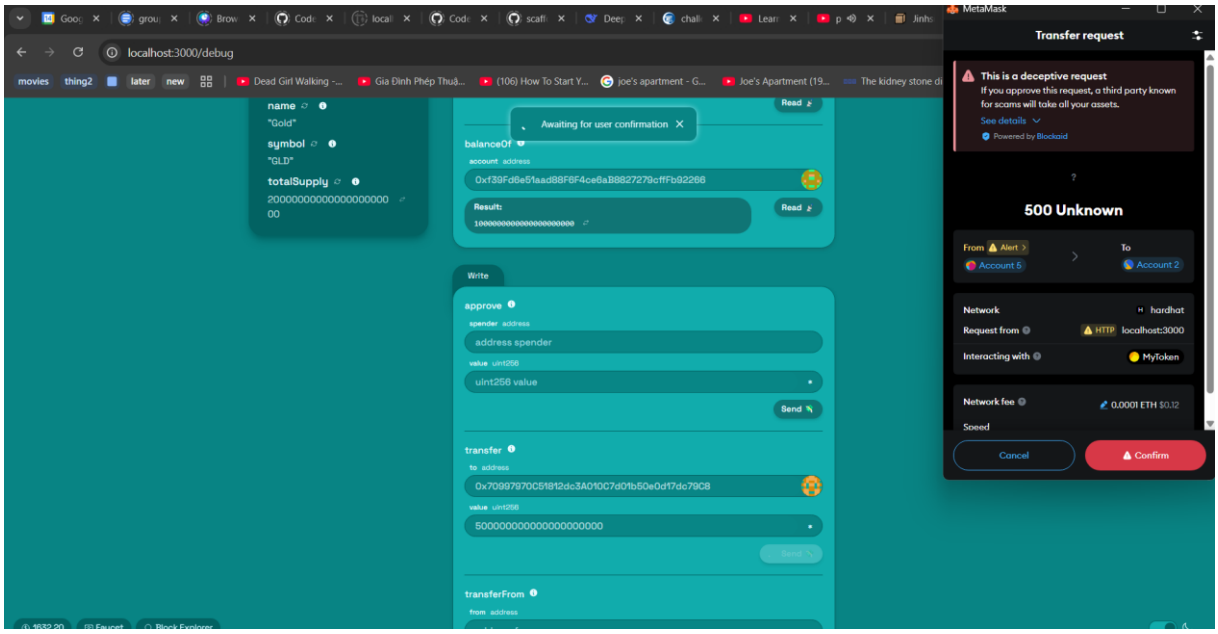


Checking token balances:

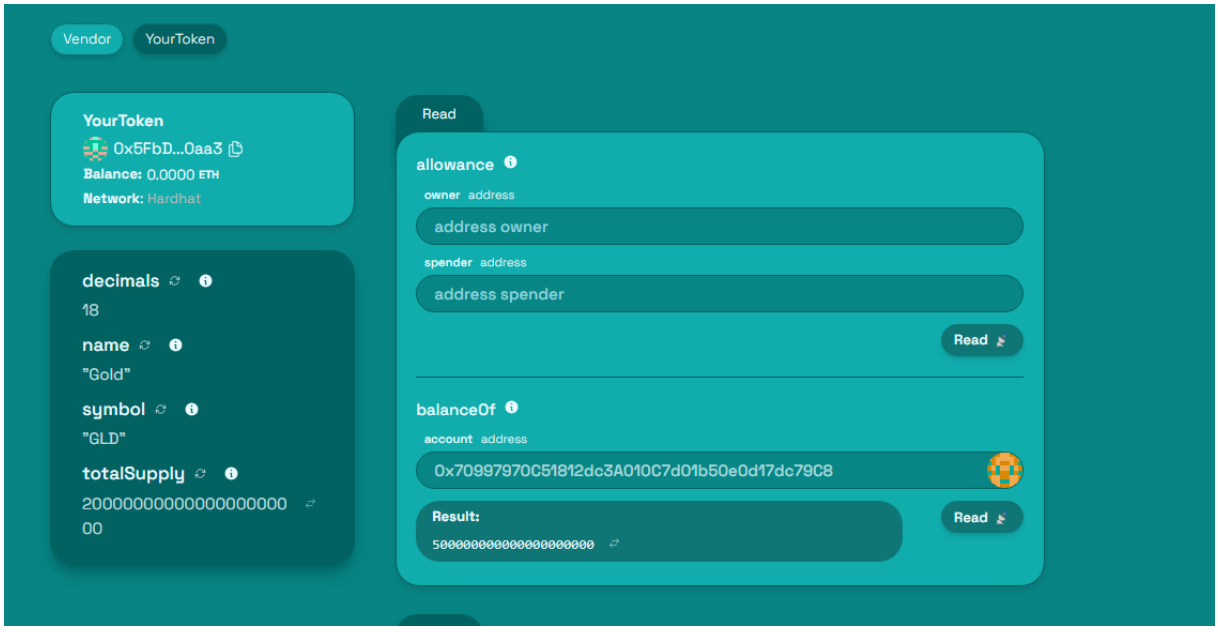


Here the number of tokens is measured in weis not full tokens, which is why it looks so large, in fact this account only has 1000 erc20 tokens, the other 1000 erc20 tokens is held by the vendor

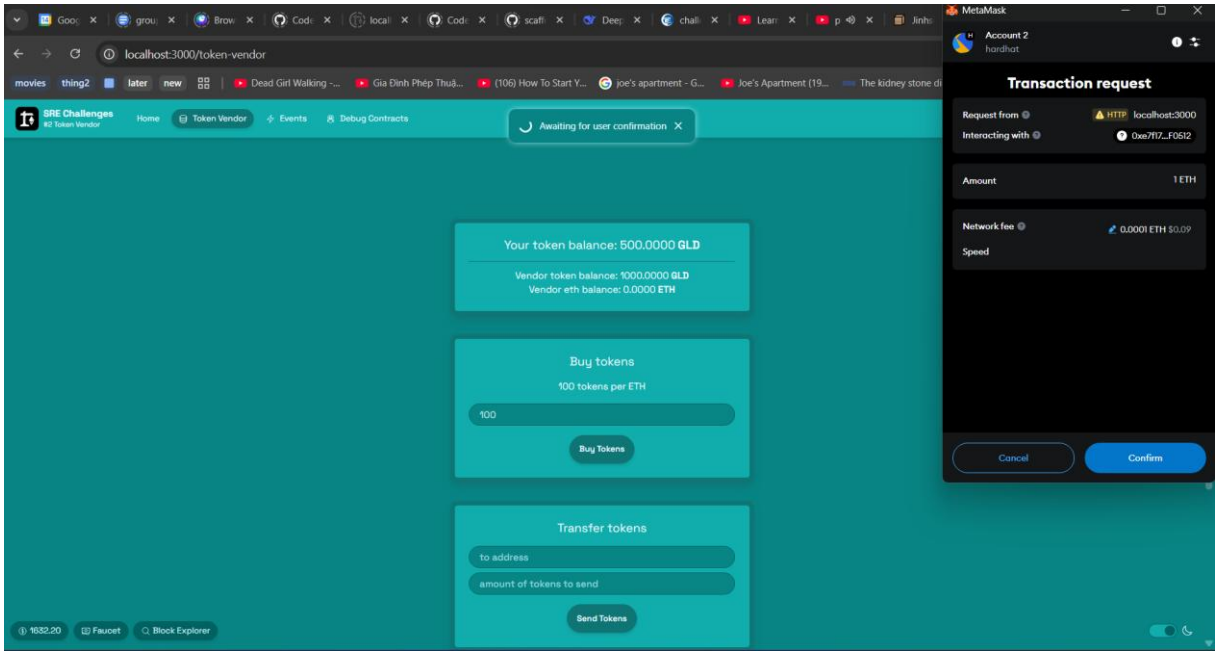
Transfering tokens:



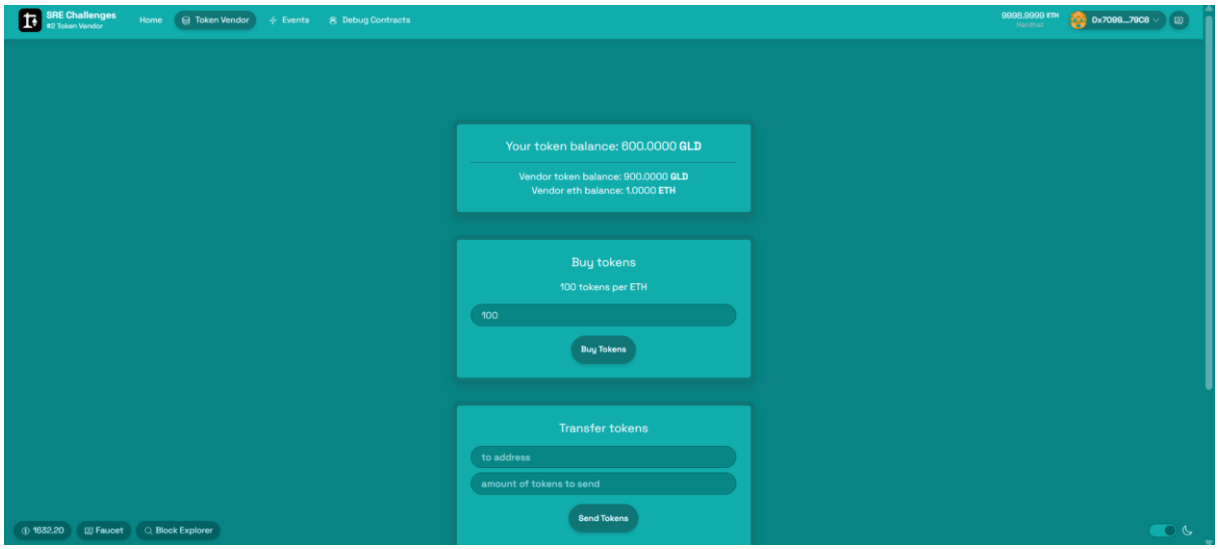
Here I tranfer 500 full tokens to another account which I verify:



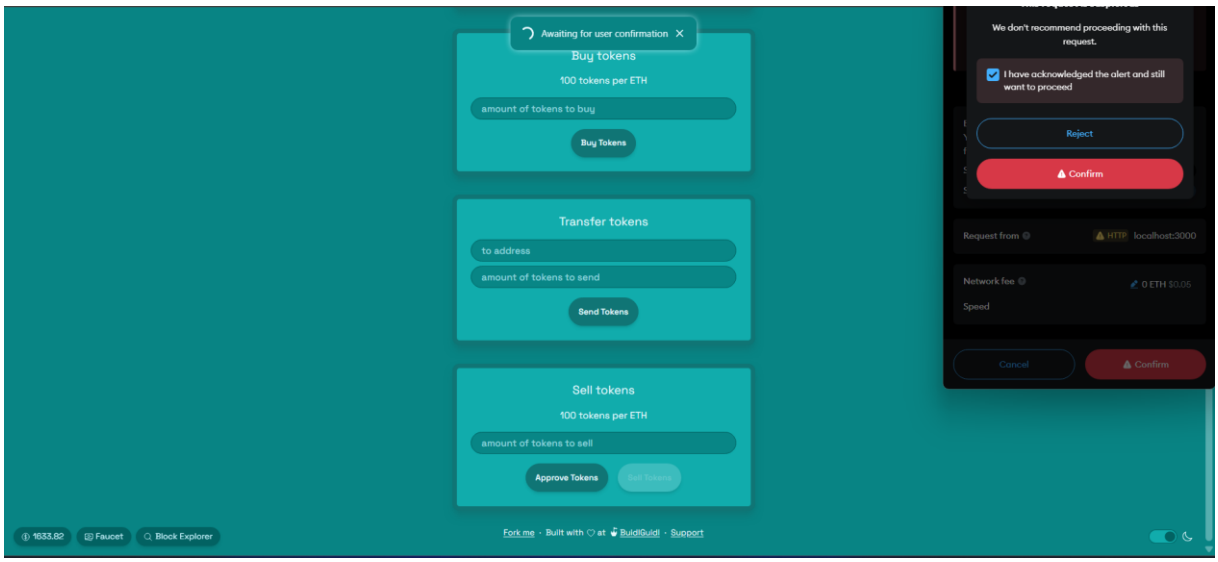
Buying and selling tokens with the vendor smart contract:



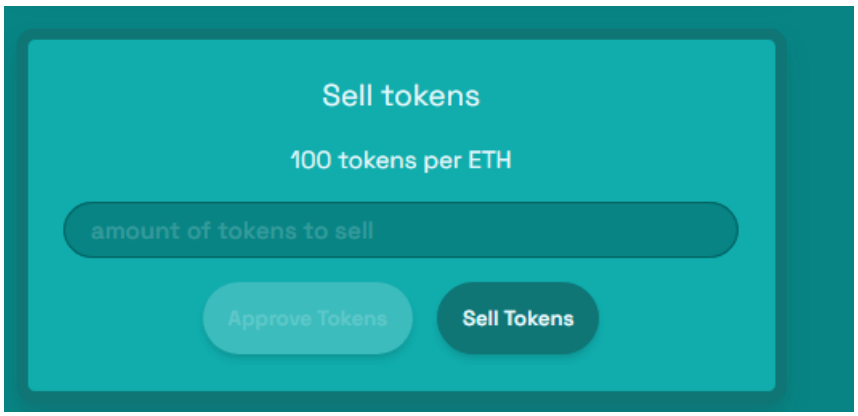
Here I buy 100 erc20 tokens for 1 eth so my balance becomes 600



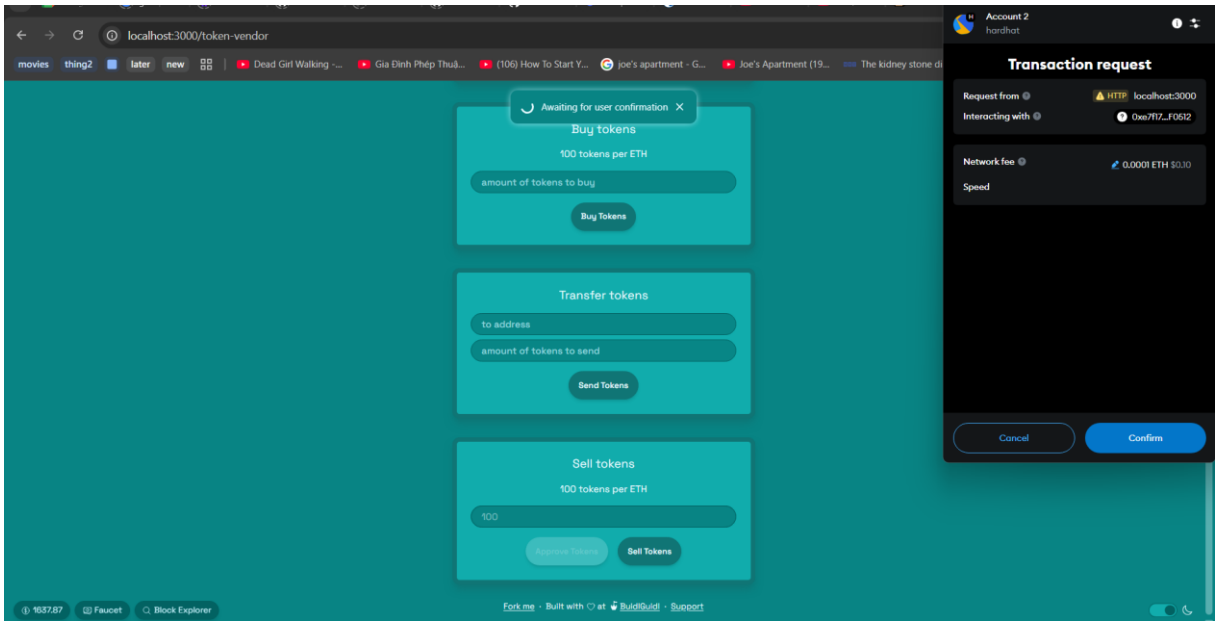
Selling erc20 tokens to get eth is more complicated, the account must approve the tokens so the token can call the transferFrom() function :

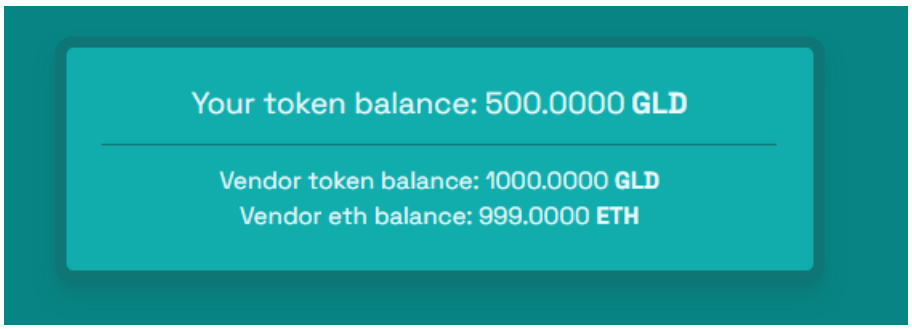


After approving:



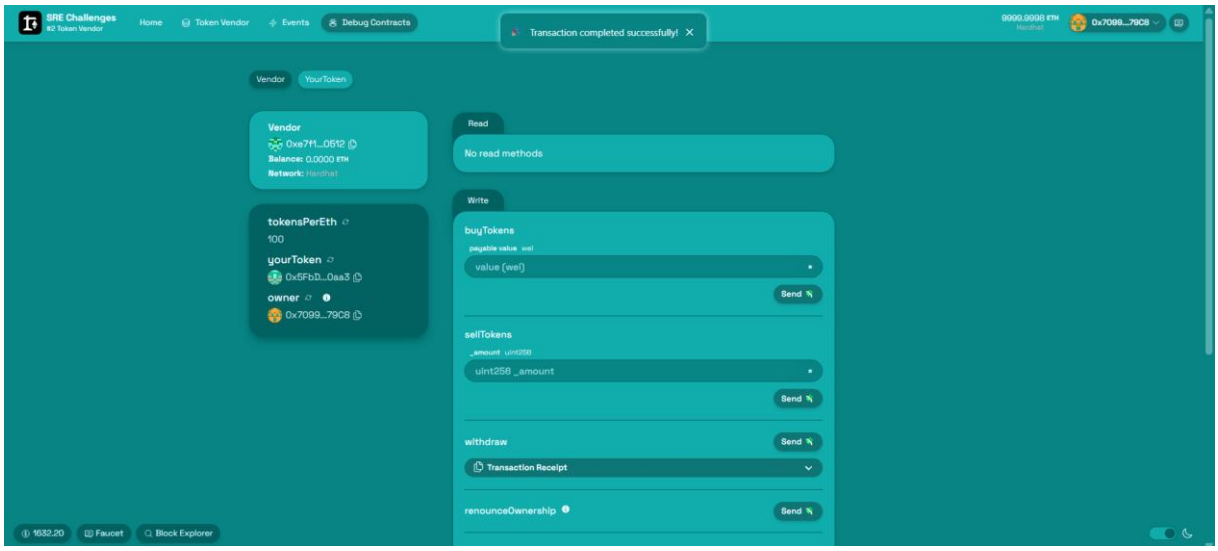
Selling the tokens after approving:



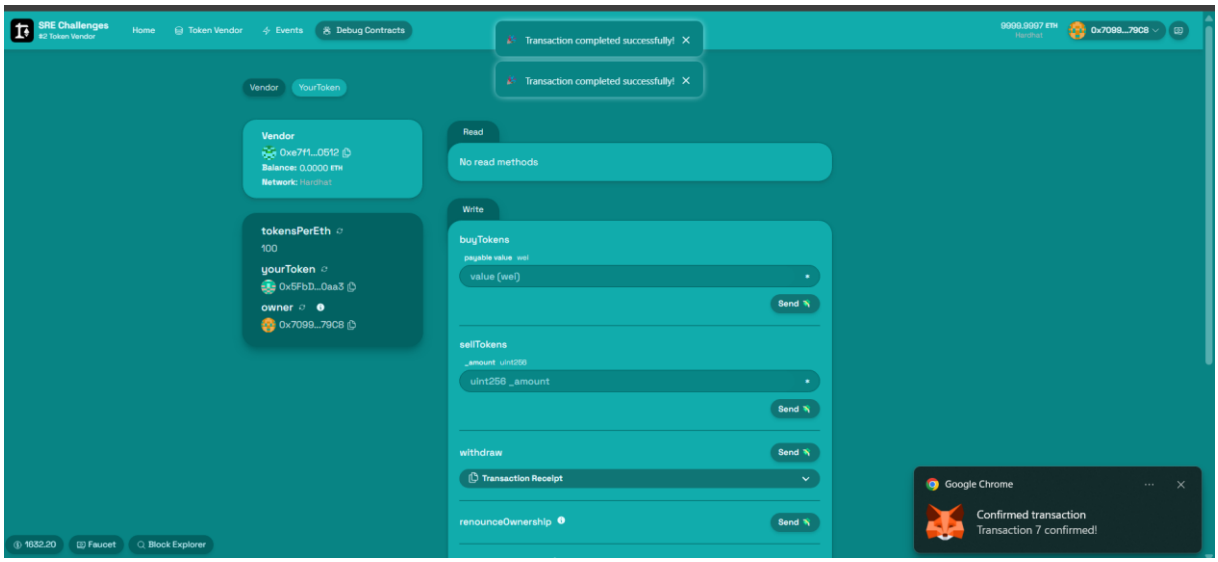


Running the withdraw function which returns all eth tokens to the owner account

Owner account and vendor balance before withdrawing:



After:



Events page which shows the events emitted by the vendor: