Insurance Smart Contract

This project develops a decentralized, transparent, automated insurance system using Ethereum blockchain technology.

1. We will create a smart contract for the Insurance policy creation, premium payment and claim processing.

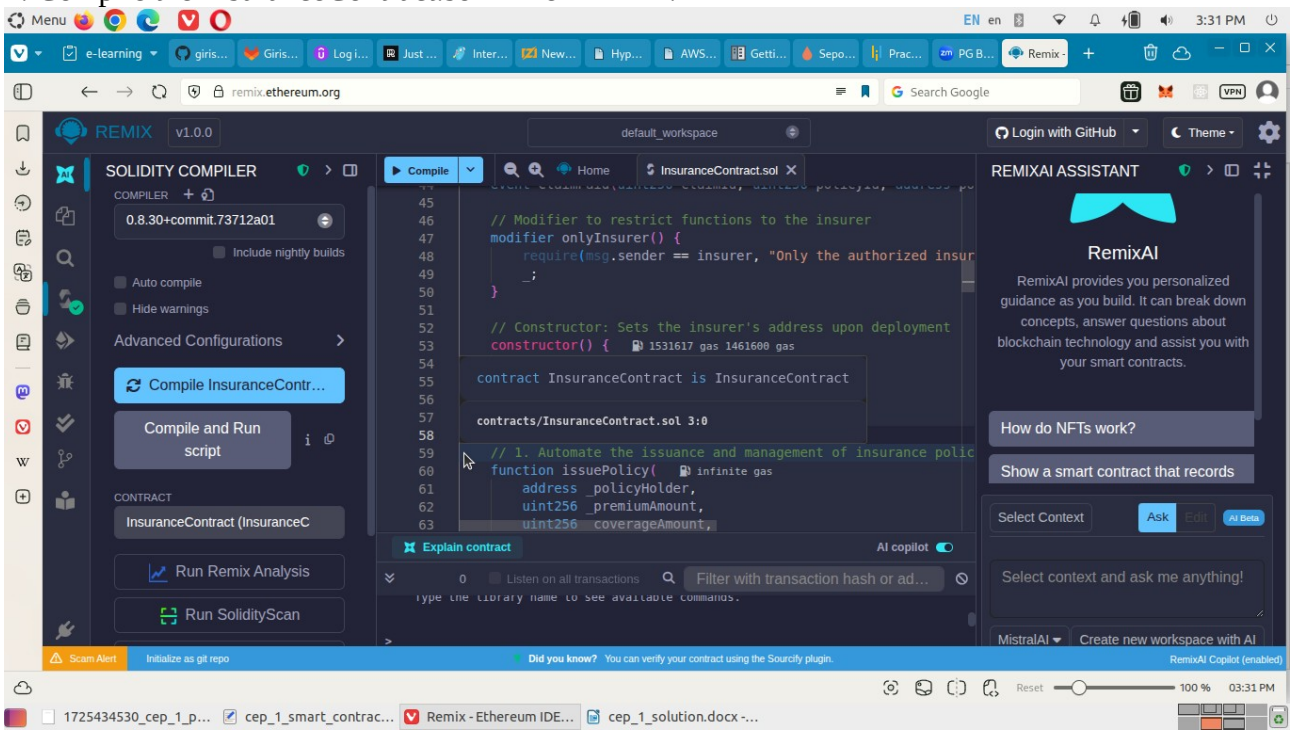
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract InsuranceContract {
    // State Variables
    address payable public insurer; // The authorized insurer's address
    uint256 private policyCounter;  // To generate unique policy IDs
    uint256 private claimCounter;   // To generate unique claim IDs

    // Enums for policy and claim status
    enum PolicyStatus { Active, Expired, Lapsed }
    enum ClaimStatus { Pending, Approved, Paid, Rejected }

    // Structs to define data structures
    struct Policy {
        uint256 policyId;
        address policyHolder;
        uint256 premiumAmount; // in Wei
        uint256 coverageAmount; // in Wei
        uint256 durationDays;
        uint256 issueTime;
        PolicyStatus status;
        uint256 lastPremiumPaymentTime;
    }

    struct Claim {
        uint256 claimId;
        uint256 policyId;
        address policyHolder;
        uint256 claimAmount; // in Wei
        string reason;
        ClaimStatus status;
    }

    // Mappings for data storage
    mapping(uint256 => Policy) public policies;
    mapping(uint256 => Claim) public claims;

    // Events for transparency and auditing (as required)
    event PolicyIssued(uint256 policyId, address policyHolder, uint256 premium);
    event PremiumPaid(uint256 policyId, address policyHolder, uint256 amount);
    event ClaimSubmitted(uint256 claimId, uint256 policyId, address policyHolder, uint256 amount);
    event ClaimApproved(uint256 claimId, uint256 policyId);
    event ClaimPaid(uint256 claimId, uint256 policyId, address policyHolder, uint256 amount);

    // Modifier to restrict functions to the insurer
    modifier onlyInsurer() {
        require(msg.sender == insurer, "Only the authorized insurer can call this function.");
        _;
    }

    // Constructor: Sets the insurer's address upon deployment
    constructor() {
        insurer = payable(msg.sender);
        policyCounter = 100;
```

```
    claimCounter = 500;
}

// 1. Automate the issuance and management of insurance policies
function issuePolicy(
    address _policyHolder,
    uint256 _premiumAmount,
    uint256 _coverageAmount,
    uint256 _durationDays
) public onlyInsurer returns (uint256) {
    // Ensure only authorized insurers can issue policies

    uint256 newPolicyId = policyCounter;
    policyCounter++;

    policies[newPolicyId] = Policy({
        policyId: newPolicyId,
        policyHolder: _policyHolder,
        premiumAmount: _premiumAmount,
        coverageAmount: _coverageAmount,
        durationDays: _durationDays,
        issueTime: block.timestamp,
        status: PolicyStatus.Active,
        lastPremiumPaymentTime: block.timestamp // Premium is assumed paid upon issuance
    });

    // Log the action for transparency
    emit PolicyIssued(newPolicyId, _policyHolder, _premiumAmount);
    return newPolicyId;
}

// Helper function to check policy status and duration
function _isPolicyActive(Policy storage policy) internal view returns (bool) {
    // Check if the current time is within the policy's duration
    bool notExpired = policy.issueTime + (policy.durationDays * 1 days) > block.timestamp;

    // Check if the policy has lapsed (for simplicity, assuming annual payment and lapsing after a grace period)
    // A more complex system would check frequency (e.g., monthly)
    bool notLapsed = policy.status == PolicyStatus.Active;

    return notExpired && notLapsed;
}

// 2. Enable Policyholders to Pay Premiums Directly Through the Smart Contract
function payPremium(uint256 _policyId) public payable {
    Policy storage policy = policies[_policyId];

    // Ensure the payment amount matches the policy's premium
    require(msg.value == policy.premiumAmount, "Payment amount must match the required premium.");

    // Ensure the policy is active before accepting payment
    require(_isPolicyActive(policy), "Policy is not active or has expired.");

    // Ensure the sender is the policyholder
    require(msg.sender == policy.policyHolder, "Only the policyholder can pay the premium.");

    // Update the policy status based on successful premium payments
    policy.lastPremiumPaymentTime = block.timestamp;
    // The funds remain in the contract's balance until claims are paid

    // Log the action for transparency
    emit PremiumPaid(_policyId, msg.sender, msg.value);
}
```

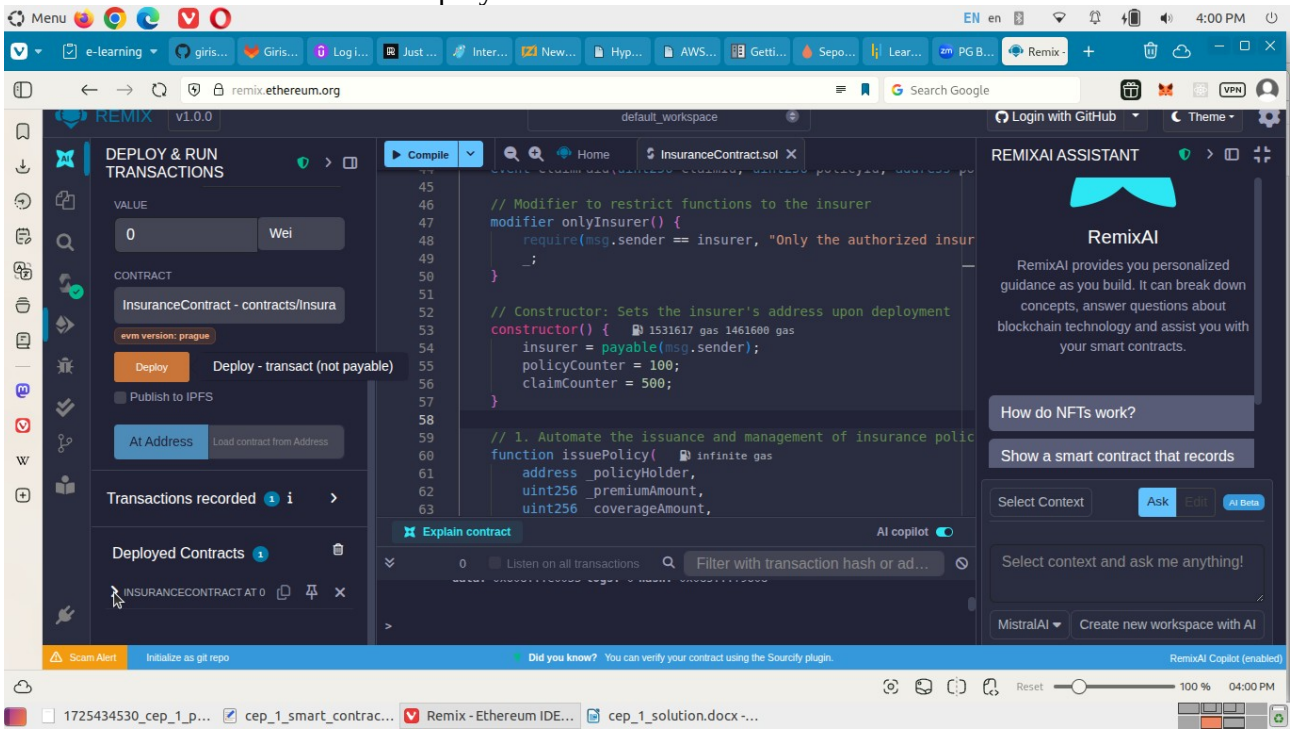
```
// 3. Allow Policyholders to Submit Claims Securely Through the Smart Contract
function submitClaim(uint256 _policyId, uint256 _claimAmount, string calldata _reason) public {
    Policy storage policy = policies[_policyId];

    // Ensure that claims are accepted only if the policy is active and not expired
    require(_isPolicyActive(policy), "Policy is not active or has expired.");

    // Ensure the sender is the policyholder
    require(msg.sender == policy.policyHolder, "Only the policyholder can submit a claim.");

    uint256 newClaimId = claimCounter;
    claimCounter++;

    claims[newClaimId] = Claim({
        claimId: newClaimId,
        policyId: _policyId,
        policyHolder: msg.sender,
        claimAmount: _claimAmount, // Record details such as the claim amount
        reason: _reason,           // Record details such as the reason for submission
        status: ClaimStatus.Pending
    });

    // Log the action for transparency
    emit ClaimSubmitted(newClaimId, _policyId, msg.sender, _claimAmount);
}

// 4. Automate the Assessment and Payout of Claims
function approveClaim(uint256 _claimId) public onlyInsurer {
    Claim storage claim = claims[_claimId];

    // Ensure the claim exists and is in a Pending state
    require(claim.status == ClaimStatus.Pending, "Claim is not in a Pending state.");

    // Allow the insurer to review and approve submitted claims
    claim.status = ClaimStatus.Approved;

    // Log the action for transparency
    emit ClaimApproved(_claimId, claim.policyId);
}

function payClaim(uint256 _claimId) public onlyInsurer {
    Claim storage claim = claims[_claimId];

    // Ensure the claim has been approved
    require(claim.status == ClaimStatus.Approved, "Claim must be Approved before payment.");

    // Ensure that each claim is approved and paid only once
    // The status check above handles this, but a safety check for contract balance is crucial.
    require(address(this).balance >= claim.claimAmount, "Contract does not have sufficient funds to pay the claim.");

    // Process approved claims, ensuring the payment is transferred to the policyholder
    // This is the core payout step (transfer in Wei)
    (bool success, ) = claim.policyHolder.call{value: claim.claimAmount}("");
    require(success, "Claim payment failed.");

    claim.status = ClaimStatus.Paid;

    // Log the action for transparency
    emit ClaimPaid(_claimId, claim.policyId, claim.policyHolder, claim.claimAmount);
}

// Fallback and Receive functions (Best practice for modern Solidity)
```

```
    // Allows contract to receive ether (premiums) through direct sends
    receive() external payable {}
    fallback() external payable {}
}
```

2. Compile the InsuranceContract.sol in Remix IDE.

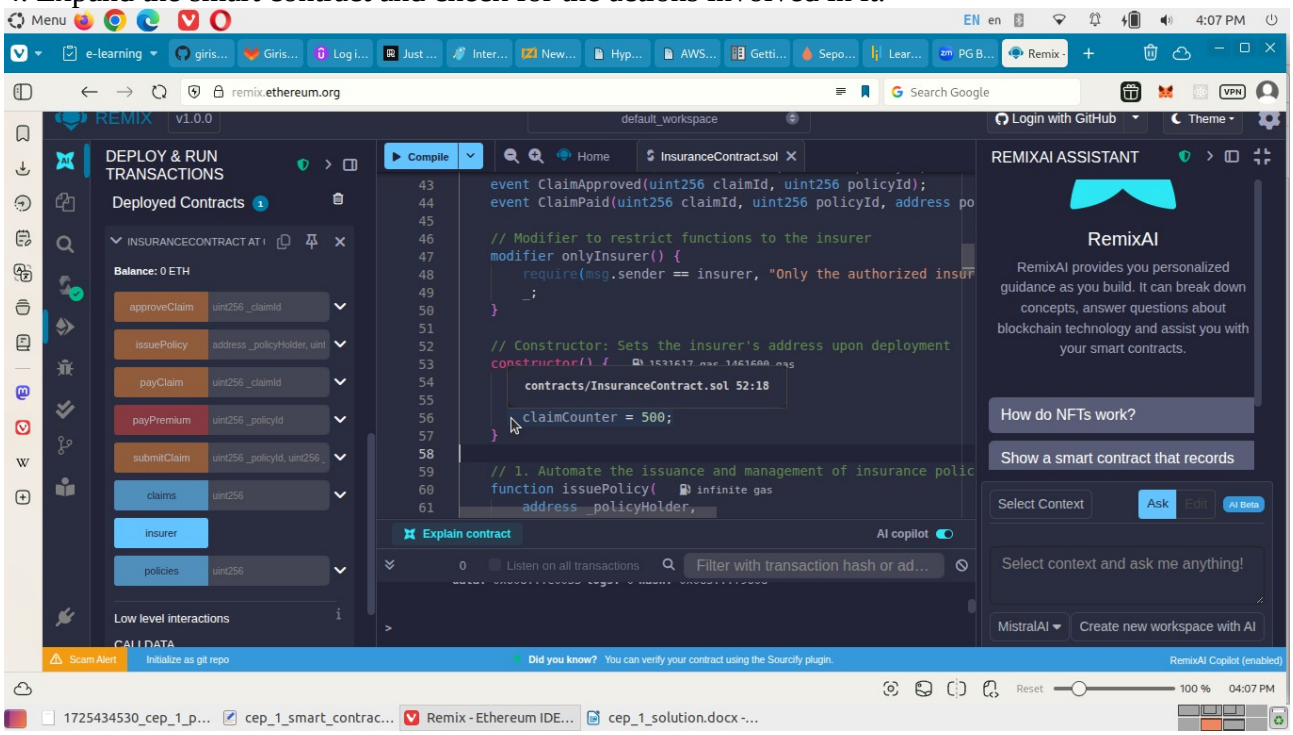


3. Deploy the transaction. Keep the Environment as Remix VM. Choose an account to start deploying the insurance smart contract. Lets select account "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4" for deploying. Select contract as "InsuranceContract". Click on Deploy button.

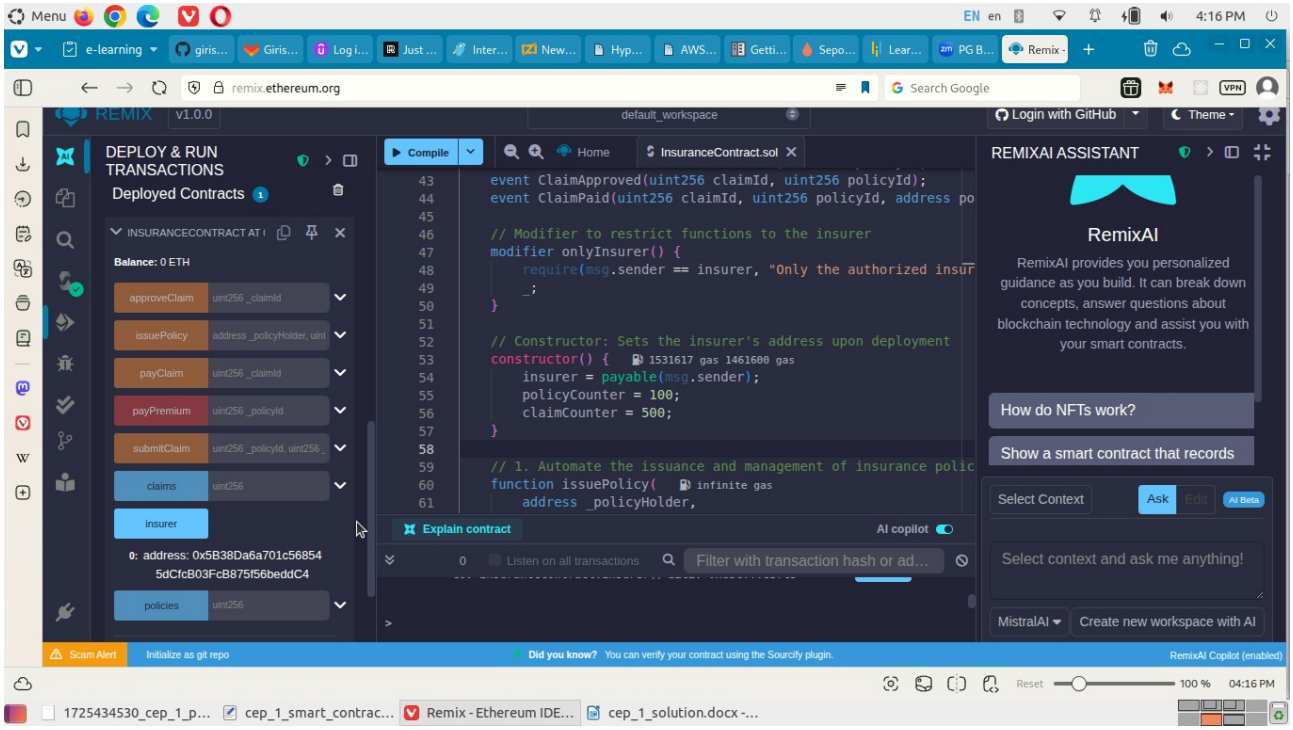


Under Deployed contracts section, you can see a smart contract has been selected. In our case, we have the address as "0x5Fcd5d9f6444dD23Ca2af792B58B041A14fB34EB".

4. Expand the smart contract and check for the actions involved in it.

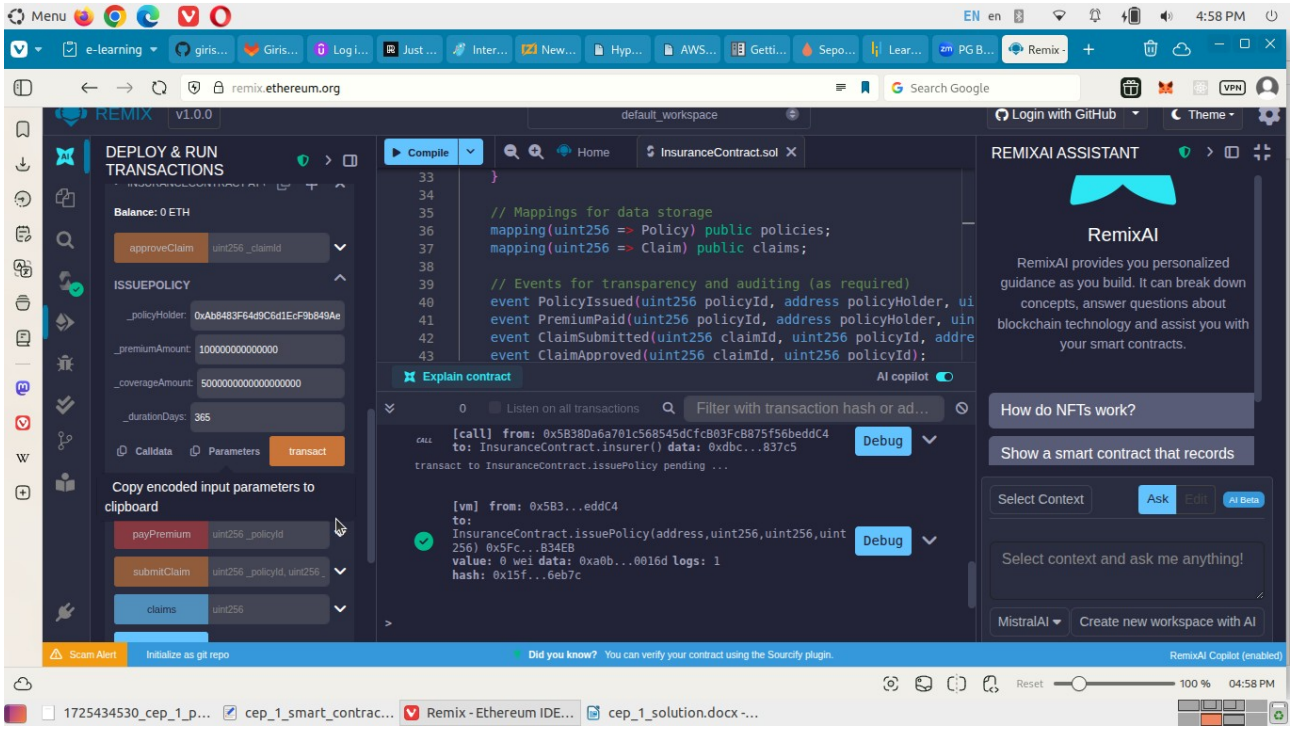


5. Click on the insurer button to see the address of the insurer.

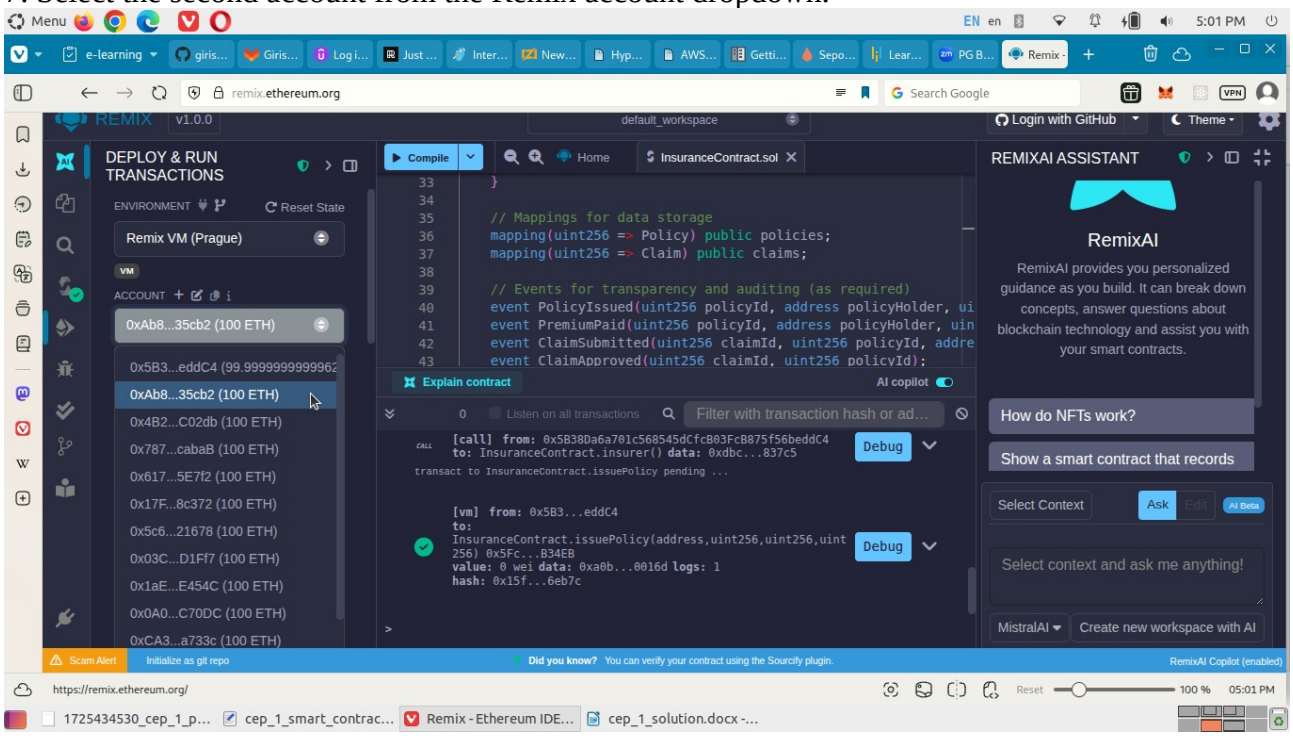


6. Now, copy address of a second account and input values as below.
_policyHolder: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
_premiumAmount: 100000000000000 (0.0001 ETH)
_coverageAmount: 5000000000000000000 (5 ETH in wei)
_durationDays: 365
Click on transact button to issue a policy to the policy holder.

P.T.O.

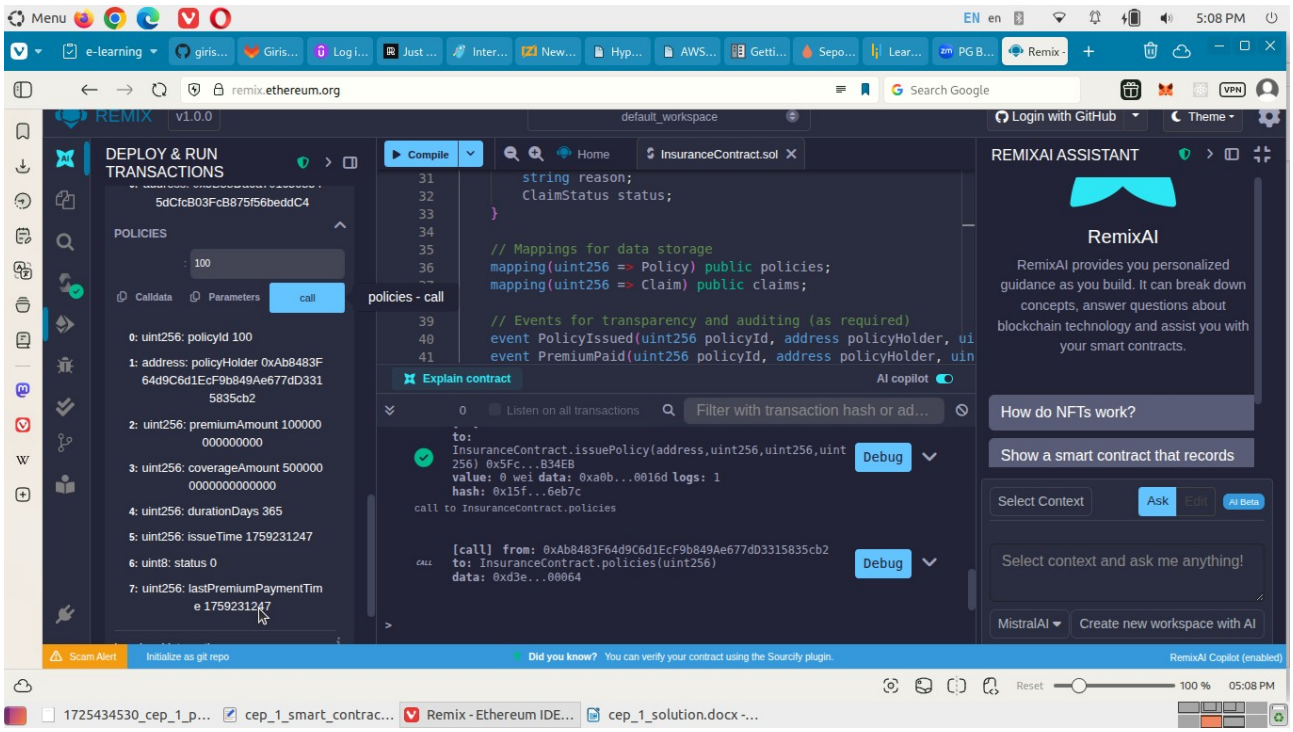


7. Select the second account from the Remix account dropdown.

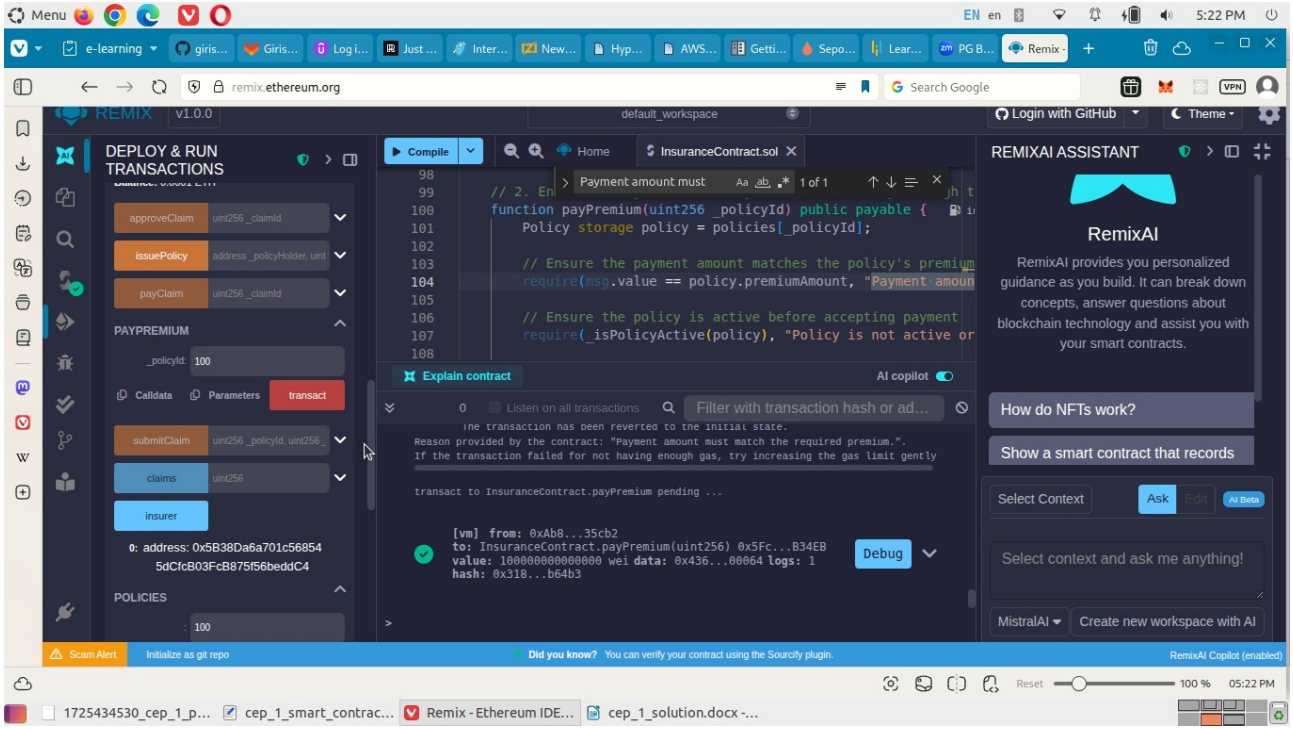


Click on policies. Supply policyId as 100. Click on call button.
Notice that, policy status is 0 i.e. Active.

P.T.O.

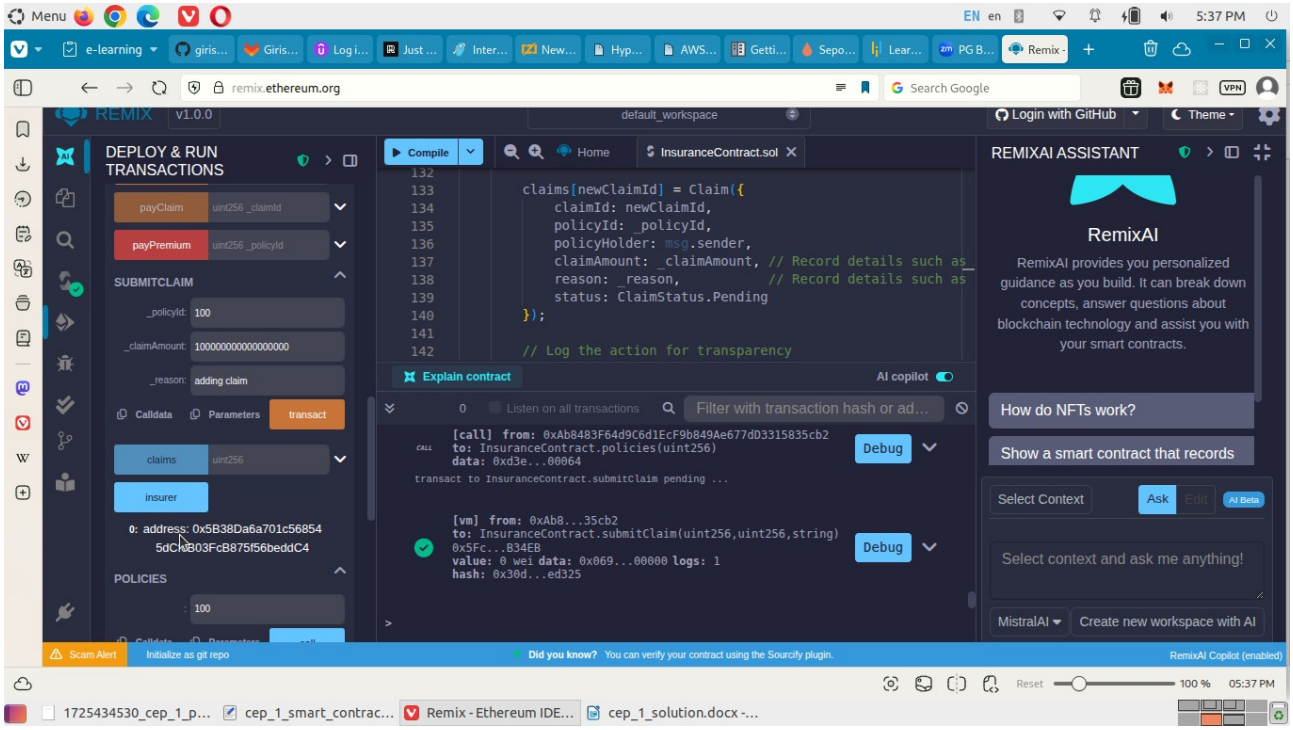


8. Expand payPremium and supply _policyId as 100, supply value as 100000000000000. Click on transact.

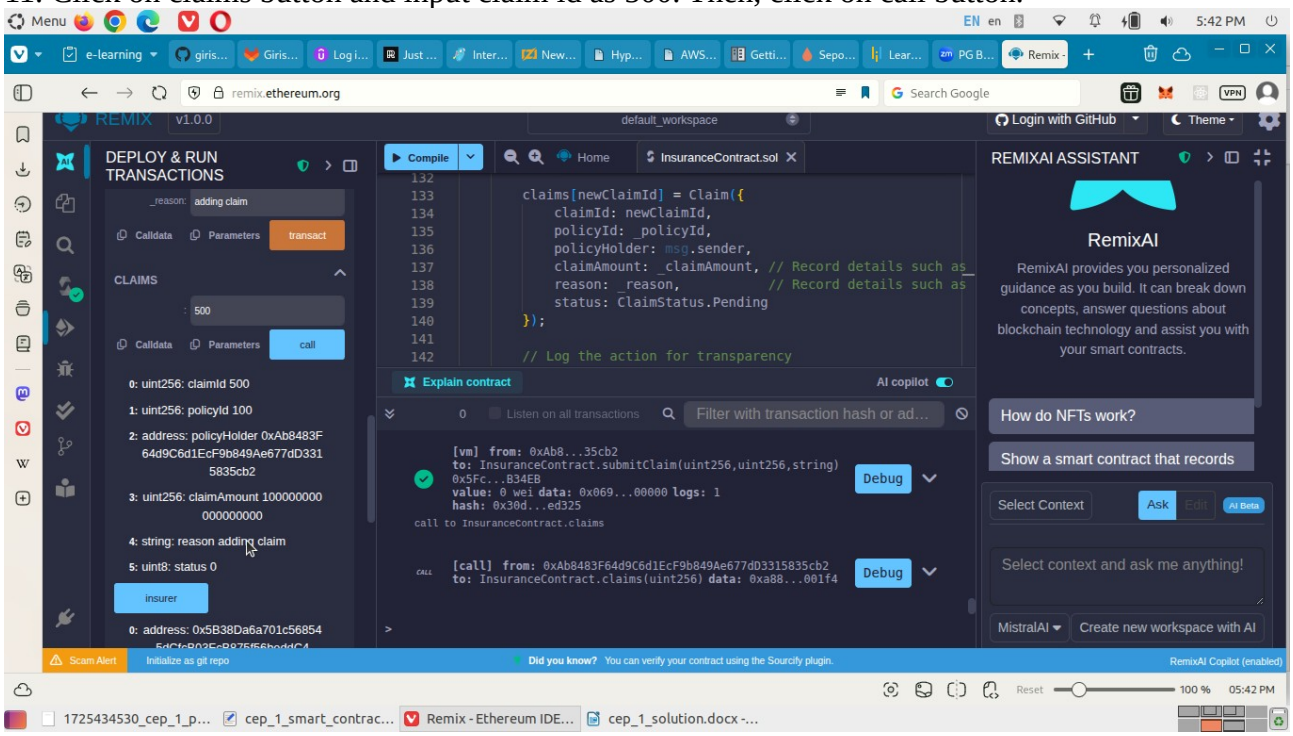


9. Click on policies and get data for policyId 100. The lastPremiumPaymentTime has changed.

10. Still in the second account (Policy Holder). Click on submitClaim and enter the below input.
_policyId: 100
_claimAmount: 100000000000000000 (0.1 ETH)
_reason: adding claim
Now, click on transact button.

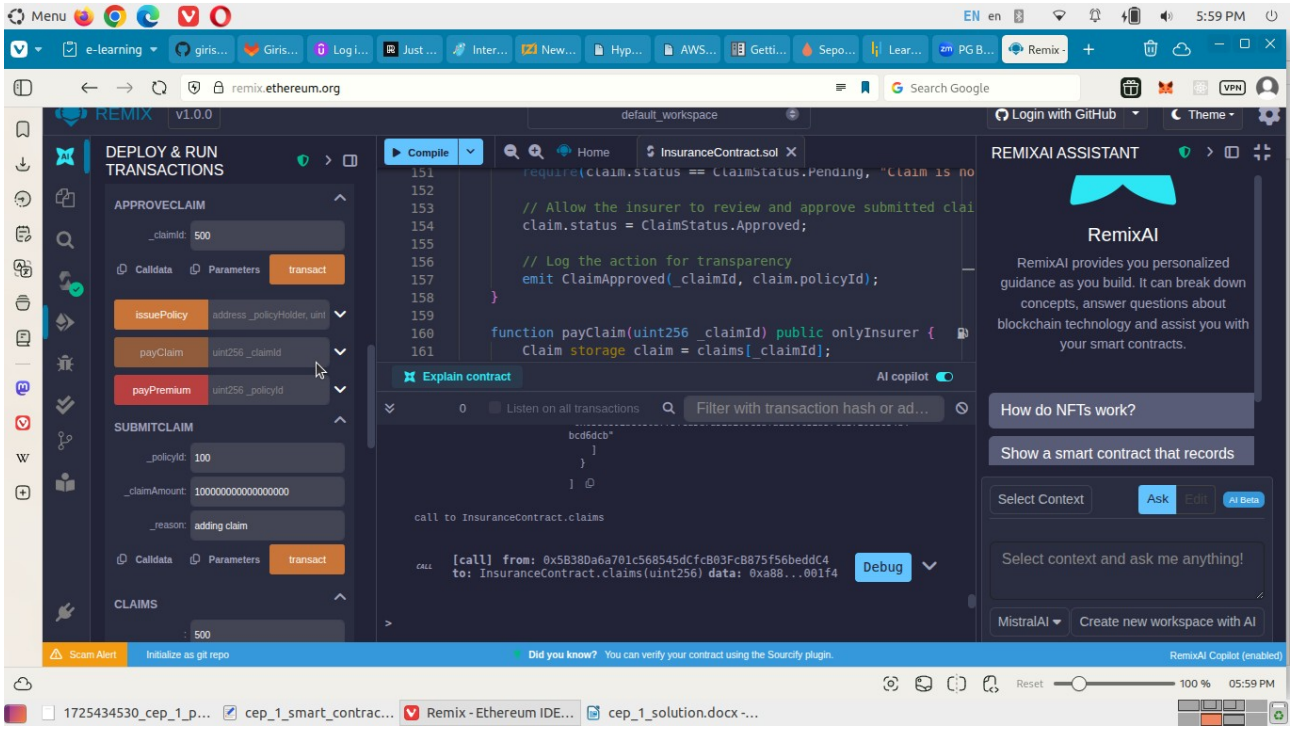


11. Click on claims button and input claim id as 500. Then, click on call button.

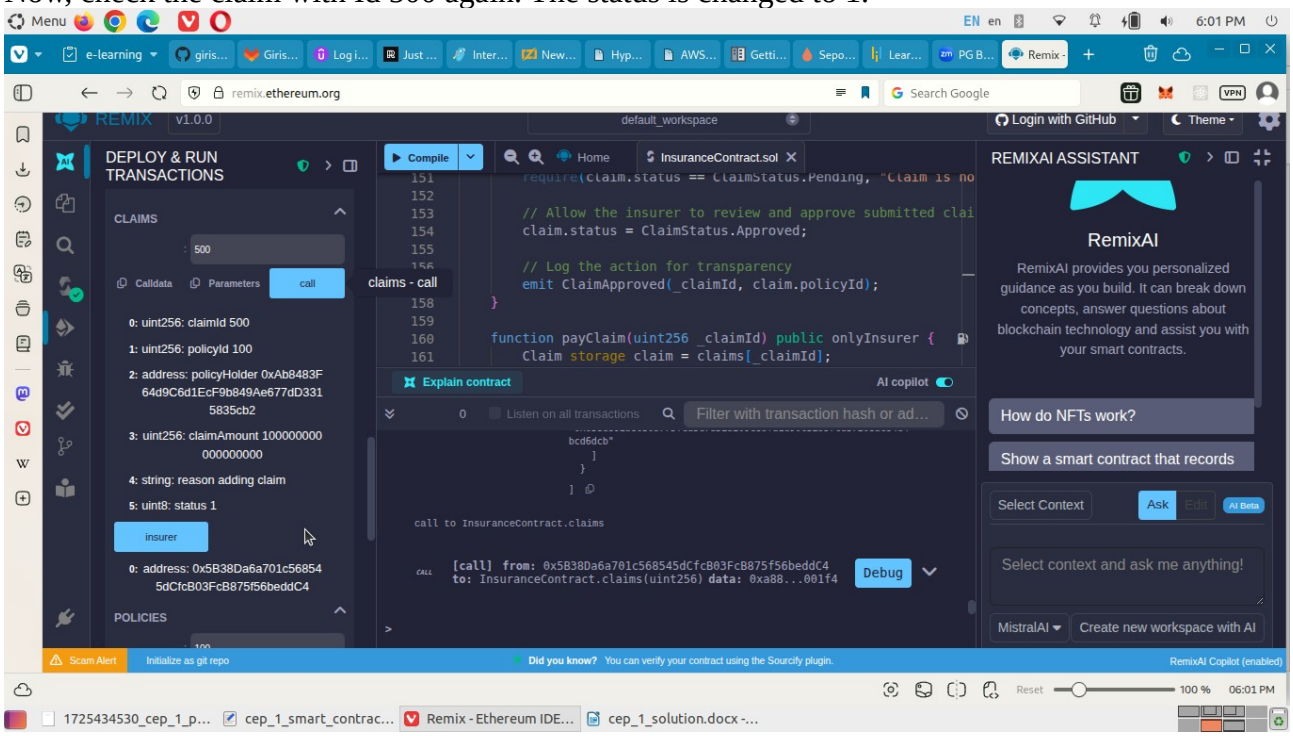


12. Switch to the first account from Remix dropdown. Account is "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4". Click on approveClaim button. Enter _claimId as 500. Then, click on transact button.

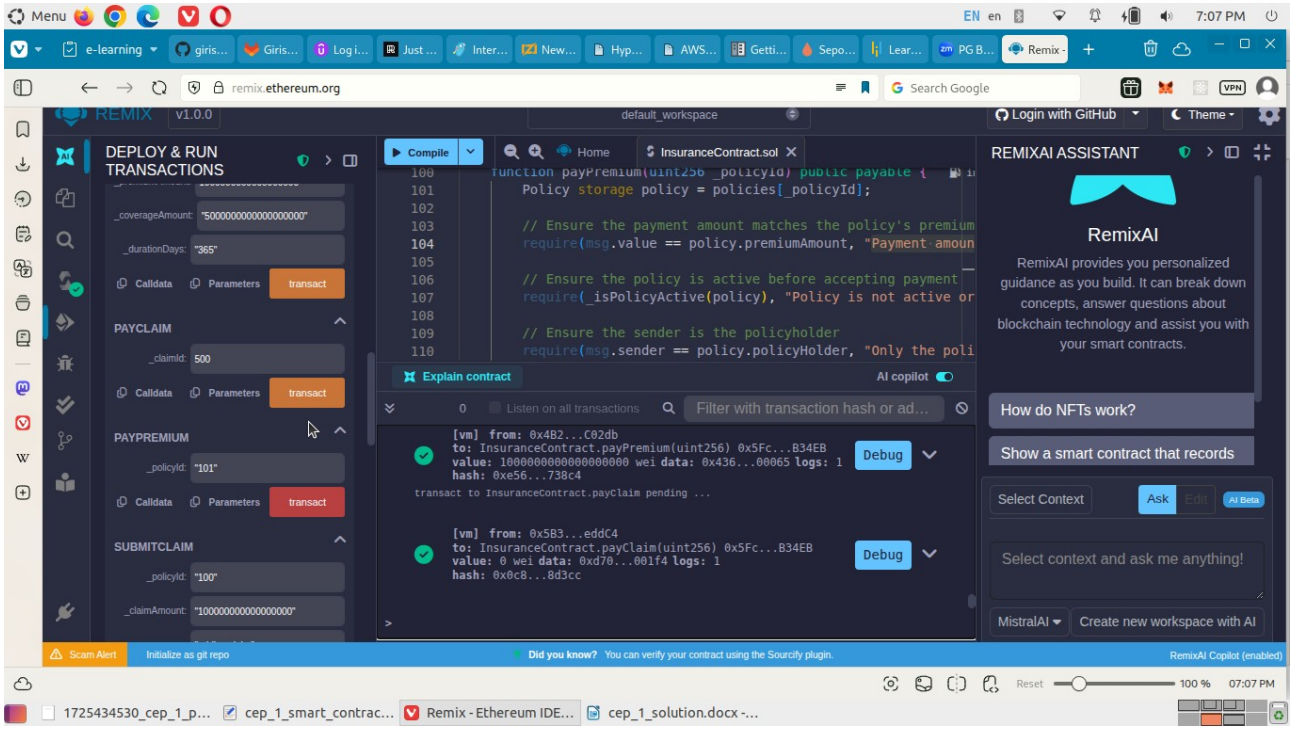
P.T.O.



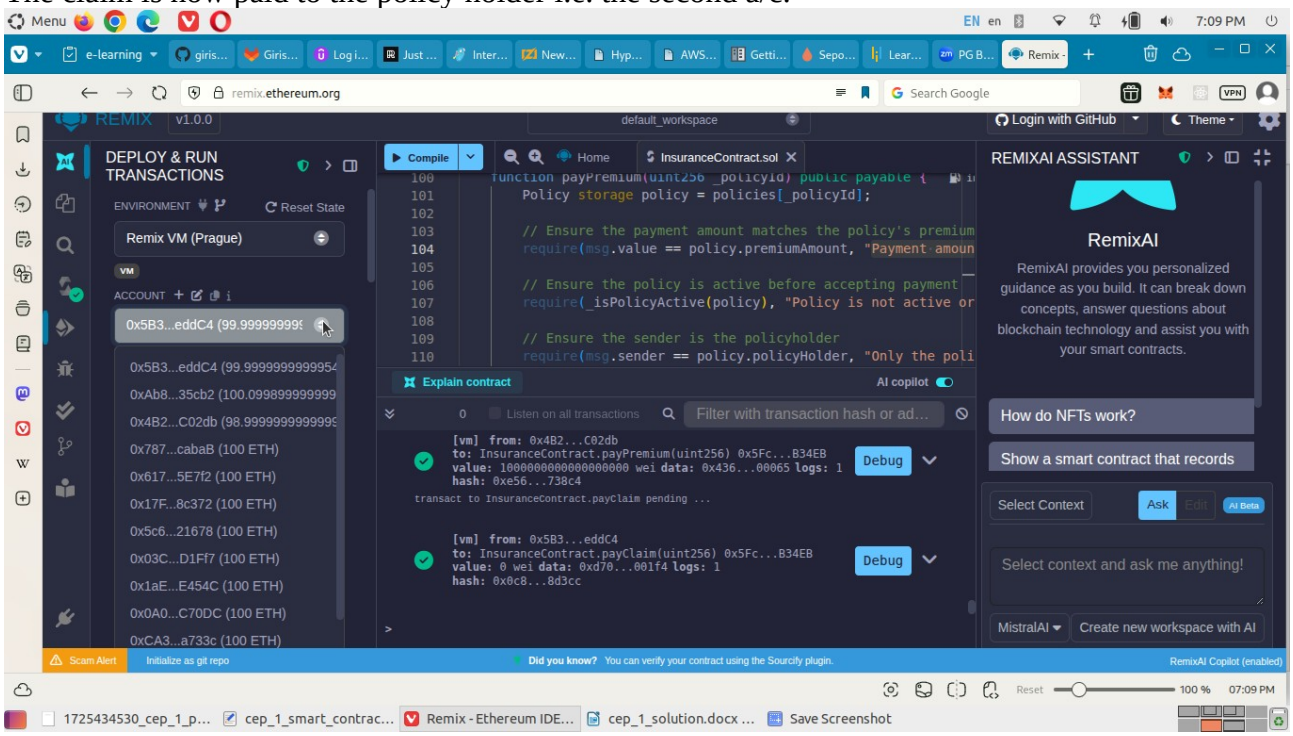
Now, check the claim with Id 500 again. The status is changed to 1.



13. Click on the payClaim button/ action and enter _claimId as 500, give value as the claim amount. Then click on transact button.



The claim is now paid to the policy holder i.e. the second a/c.



Hence, the insurance claim has been settled.