Hands On Lab With

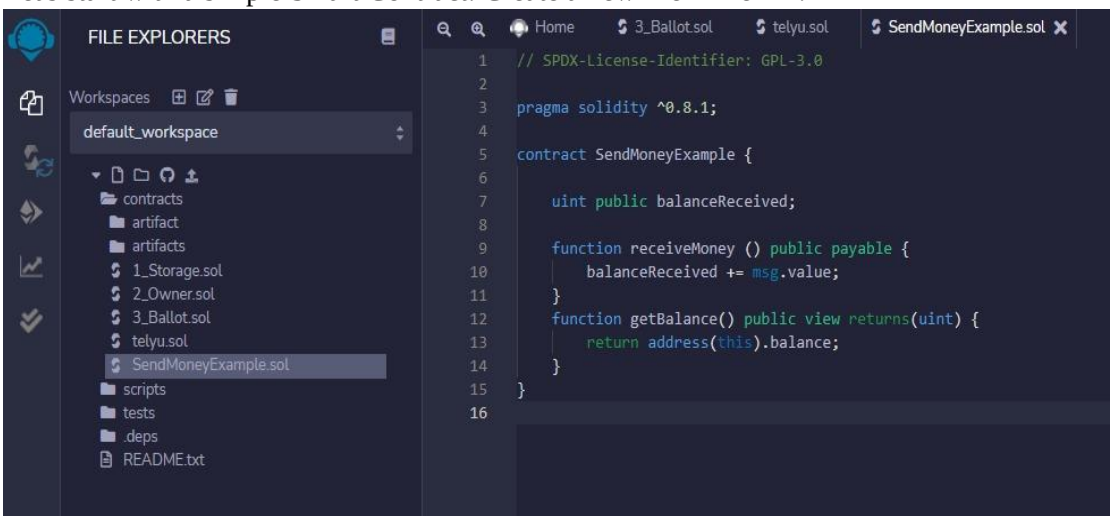
ETHEREUM BLOCKCHAIN DEVELOPER GUIDE

Delatifa Putri Sugandi
1103194080
TELKOM UNIVERSITY

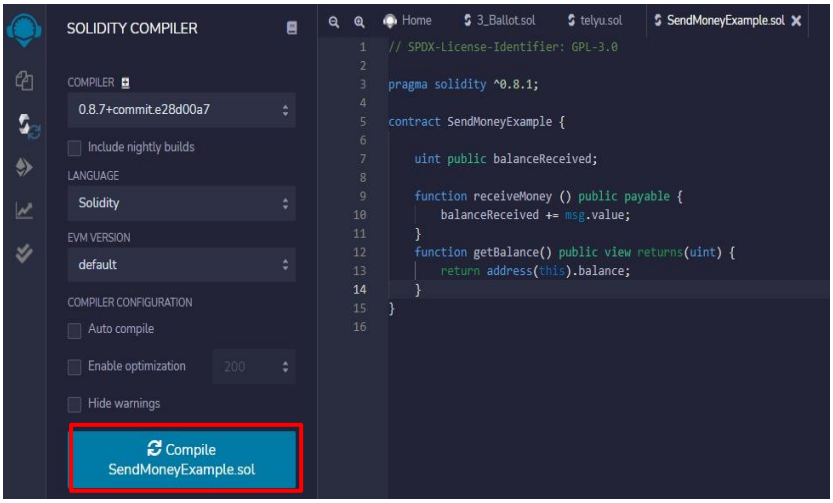
LAB 1 : DEPOSIT / WITHDRAW ETHER

- **Smart Contract**

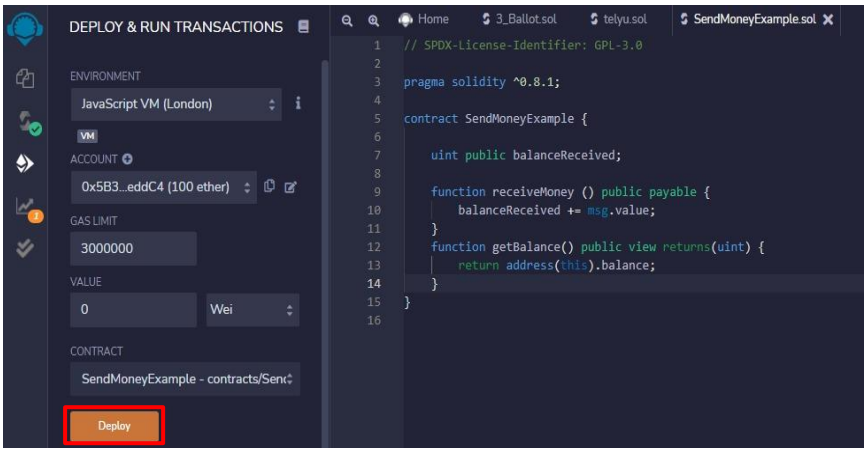
  In this lab we're going to learn how to Smart Contract manages its own funds. You will send Ether to your Smart Contract. Then the Smart Contract will manage its own Ether and will be able to relay it to anyone else. It's like a bank account with programming code attached to it.
  Let's start with a simple Smart Contract. Create a new file in Remix:



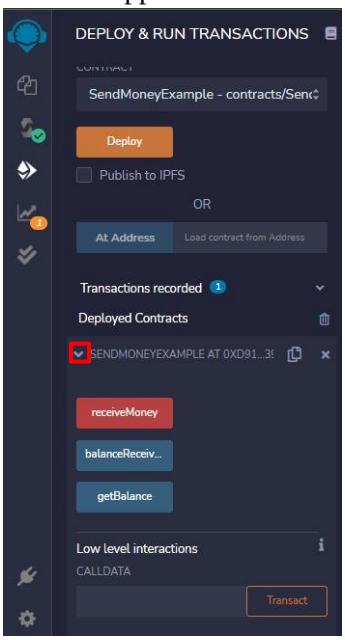
uint public balanceReceived : is a public storage variable. A public variable will create a getter function automatically in Solidity. So we can always query the current content of this variable.
balanceReceived += msg.value : The msg-object is a global always-existing object containing a few informations about the ongoing transaction. The two most important properties are .value and .sender. Former contains the amount of Wei that was sent to the smart contract. Latter contains the address that called the Smart Contract. We will use this extensively later on, so, just keep going for now.
function getBalance() public view returns(uint) : a view function is a function that doesn't alter the storage (read-only) and can return information. It doesn't need to be mined and it is virtually free of charge.
address(this).balance : A variable of the type address always has a property called.balance which gives you the amount of ether stored on that address. It doesn't mean you can access them, it just tells you how much is stored there. Remember, it's all public information. address(this) converts the Smart Contract instance to an address.
After create the new file, dont forget to complie the new file.



- **Deploy the Smart Contract**



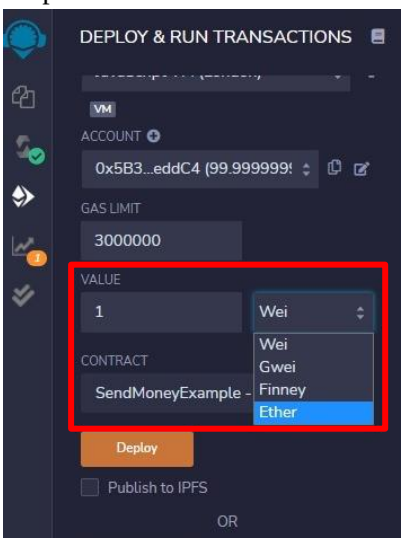
It should appear at the bottom of the Plugin - you probably need to expand the contract instance:



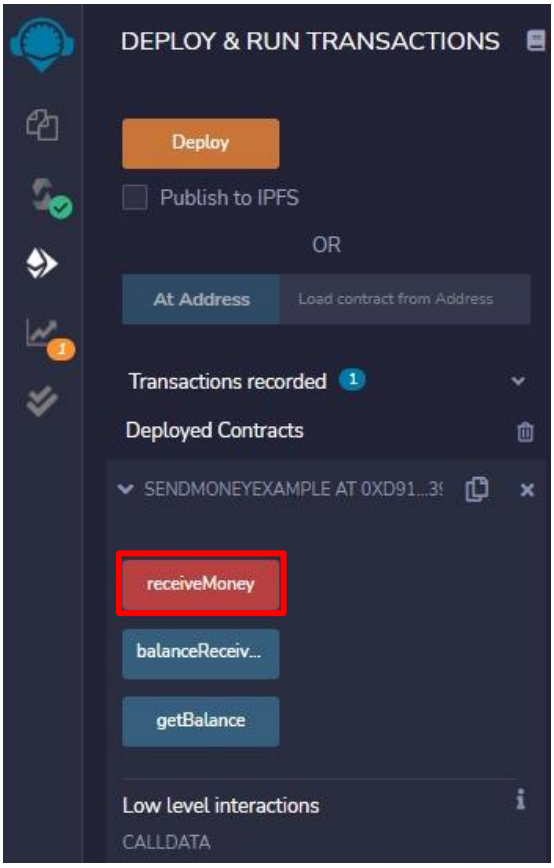
- **Send Ether to the Smart Contract**

  Now it is time to send some Ether to the Smart Contract!

  Scroll up to the "value" field and put "1" into the value input field and select "ether" from the dropdown:

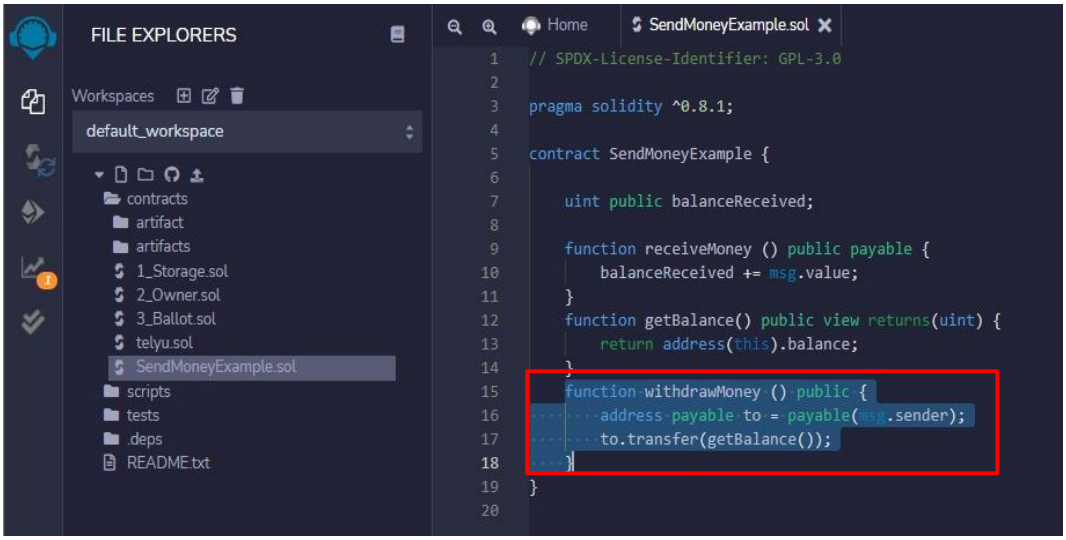


Then scroll down to the Smart Contract and hit the red "receiveMoney" button:

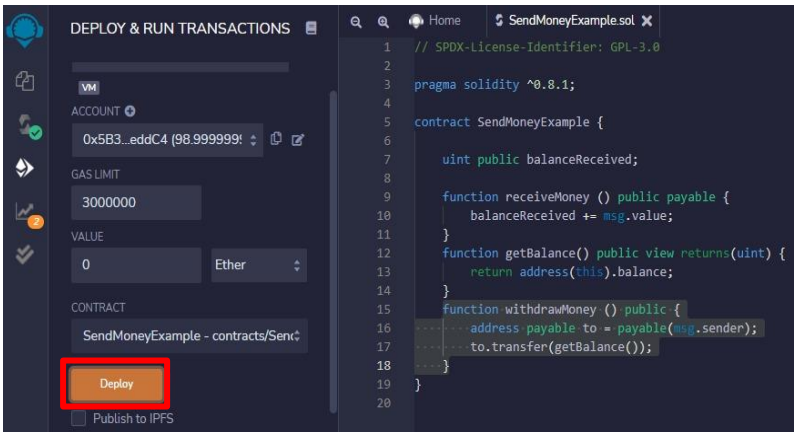


- **Check the Balance**
  Now we sent 1 Ether to the Smart Contract. According to our code the variable **balanceReceived** and the **function getBalance()** should have the same value.

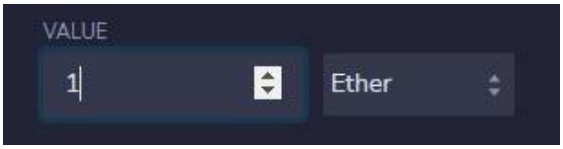
- **Withdraw Ether From Smart Contract**
  So far we have sent Ether to our Smart Contract. But there is currently no way to get Ether back out again! So, what's next? Yes! A function to withdraw Ether.
  Add the withdraw function.

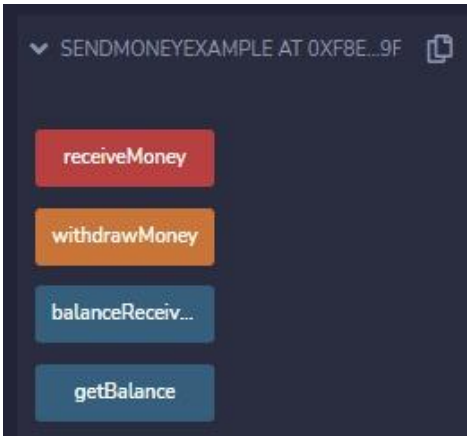


  This function will send all funds stored in the Smart Contract to the person who calls the "withdrawMoney()" function.
  After Add withdrawMoney(), you should Compile the new Smart Contract.
  Then Deploy the new version and send again 1 Ether to the Smart Contract.



  At the end you should end up with one active Instance of your Smart Contract.
  The same procedure as before:
  1. Put in "1 Ether" into the value input box
  2. Hit "receiveMoney" in your new contract Instance

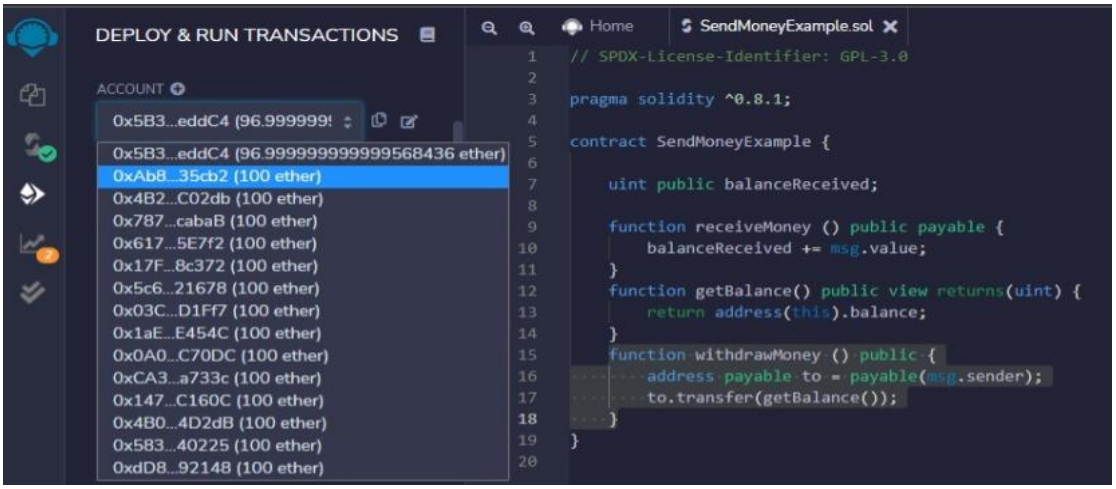




Your balance should be 1 Ether again.

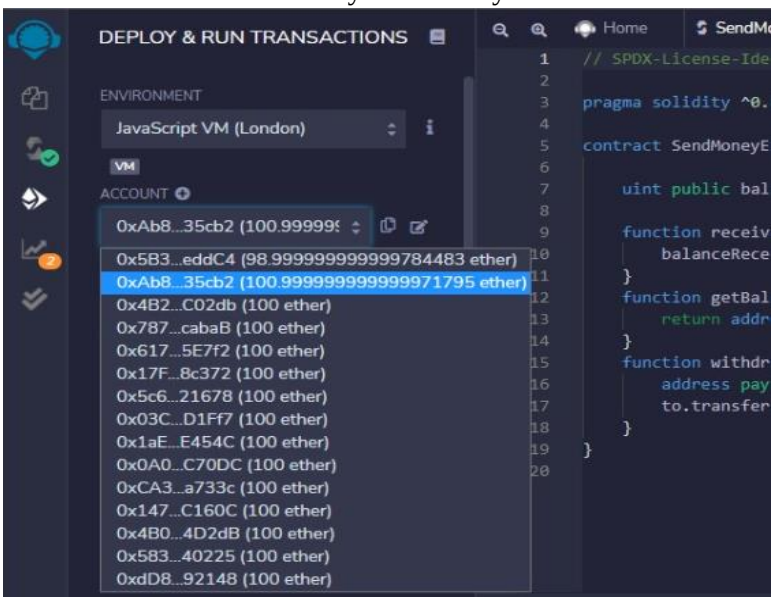
- **Withdraw Funds from the Smart Contract**

  Now it's time we use our new function! But to make things more exciting, we're going to withdraw to a different Account. Select the second Account from the Accounts dropdown:



Then hit the "withdrawMoney" button:

Observe the amount to Ether you have in your account.



It's more than the previous 100 Ether! We got our 1 Ether through our Smart Contract into another Account! AWESOME!

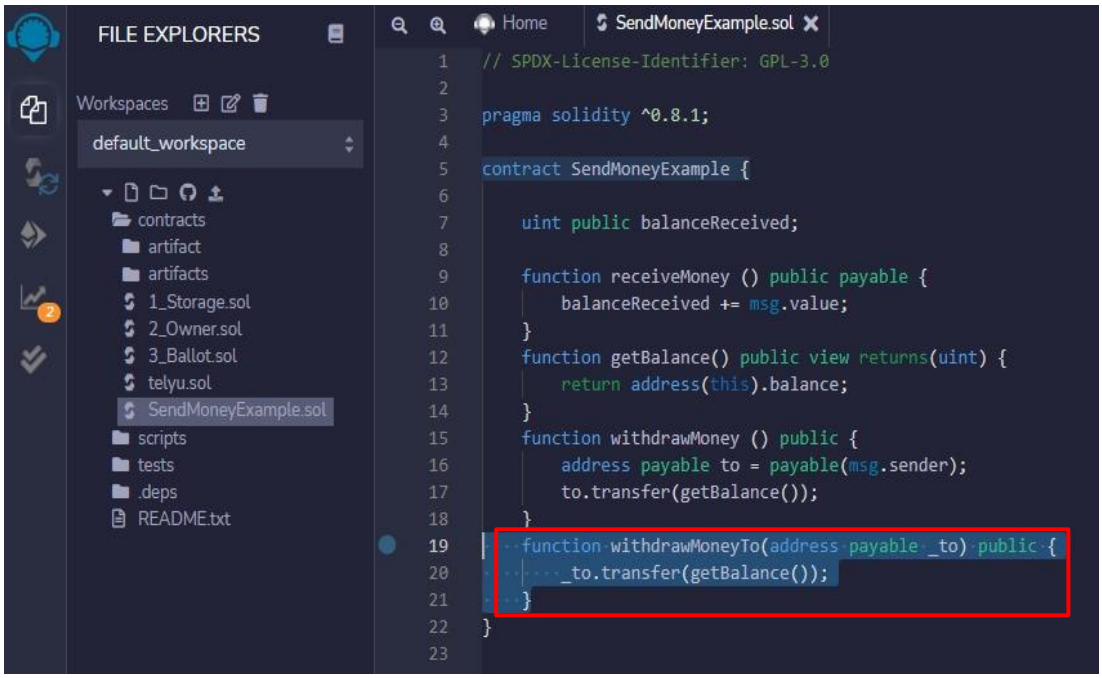
Why not 101 Ether?

Are you wondering why you don't have 101 Ether in your Account? After all, you had 100 Ether before, and now you added 1 Ether, so, why is it not 101 Ether? Is the Math you learned in school worthless? No, the Math you learned in School comes in handy actually.

What you can observe here is the concept of "Gas" on the Ethereum Blockchain. Every transaction on Ethereum costs a little bit. And it's not different here on a simulated chain. Same principles apply. How much is the Gas you paid, you're wondering? Well, you can open the transaction details and see for yourself. We're covering this - in depth - later on in the course. I also made a dedicated video and blog post about this if you want to deep dive right now.

While we can withdraw our funds now, the whole function itself is pretty useless, isn't it?! Anyone can withdraw funds to his Account. There are no fractions of the Amount - all in all, pretty insecure.

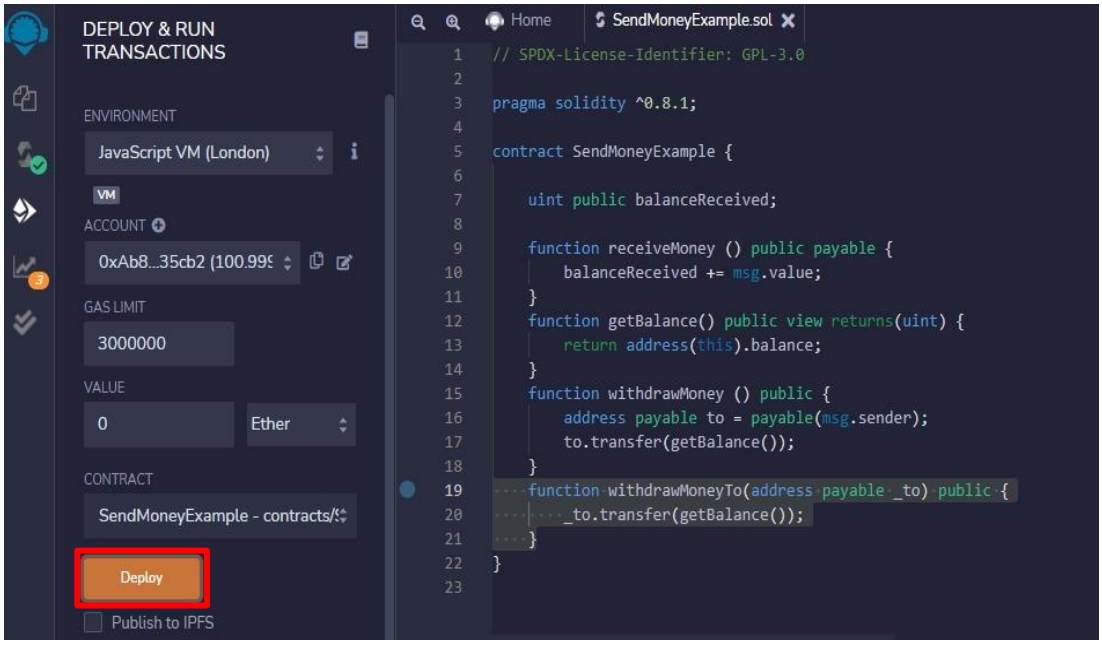
- **Withdraw to Specific Account**
  Previously we had our Smart Contract just blindly send the Ether to whoever called the Smart Contracts "withdrawMoney" function. Let's extend this a bit so that the Funds can be send to a specific Account.

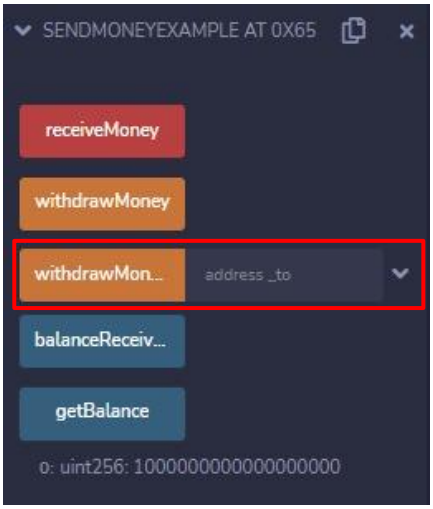


As you can see, we can now specify an Address the money will be transferred to! Let's give this a try!

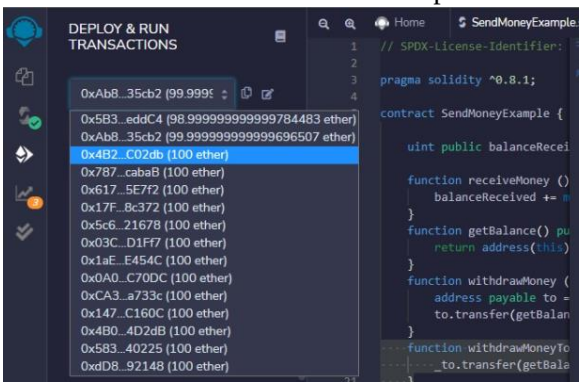
Of course, we need to re-deploy our Smart Contract. Same procedure as before:

1. Compile and Deploy the Smart Contract
2. Close the old Instance
3. Send 1 Ether to the Smart Contract (don't forget the value input field!)
4. Make sure the Balance shows up correctly.

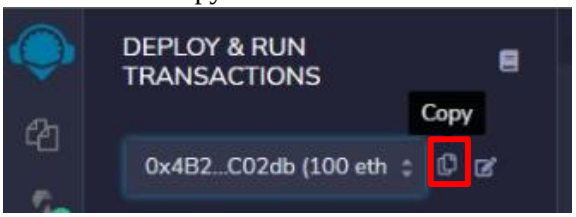




- **Test the "withdrawMoneyTo" function**
  Now it's time to test the new function. We're going to use our fist account to send all funds to the third account.
  Select the third account from the dropdown



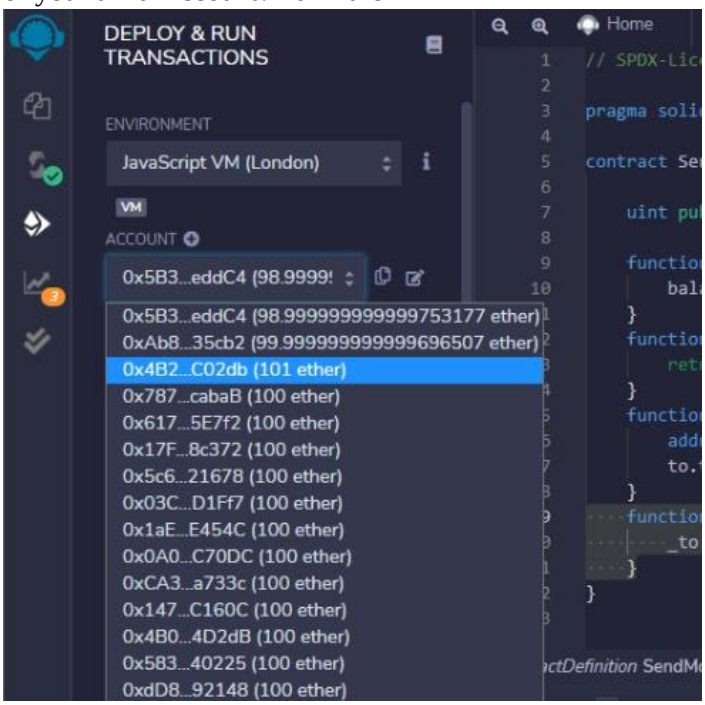
Hit the little coopy icon.



Switch back to the first Account.

Paste the Account you copied into the input field next to “withdrawMoneyTo”.

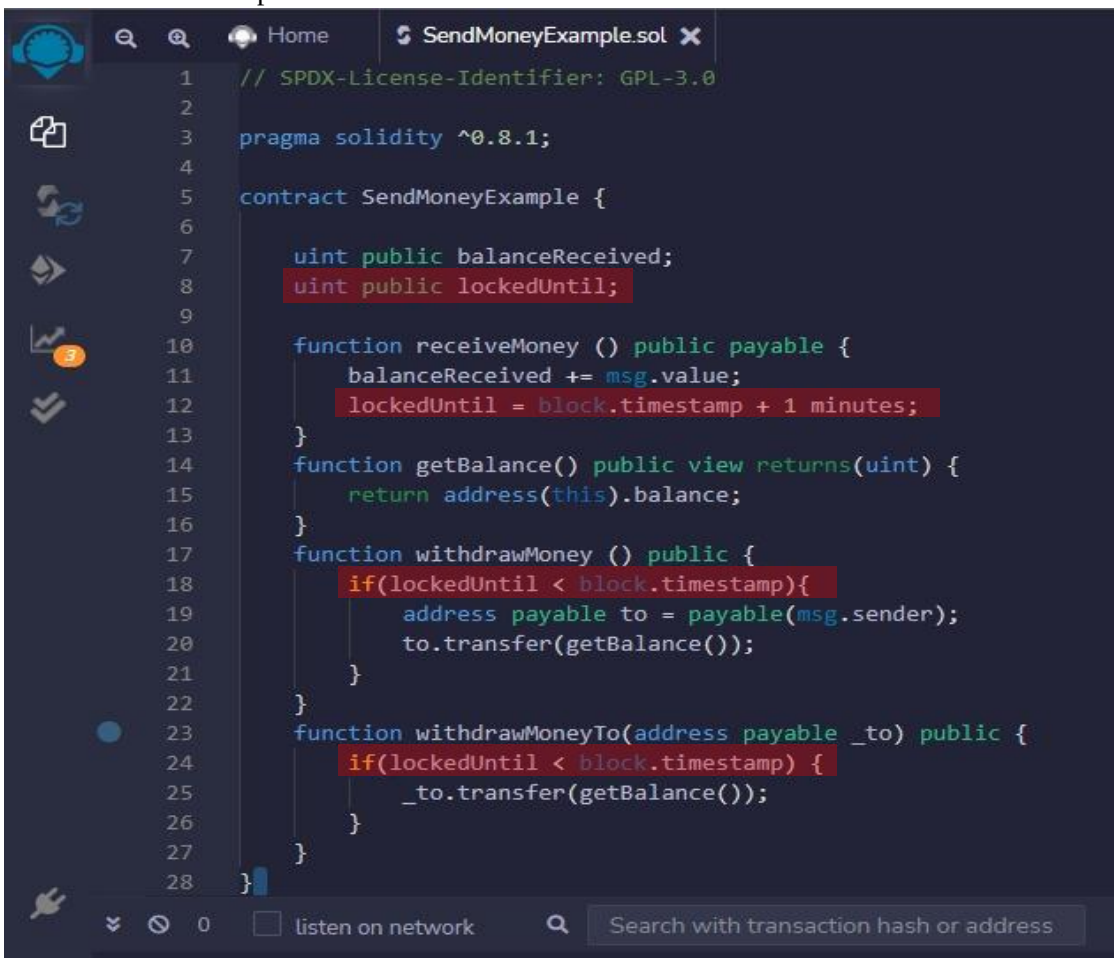
Hit the “withdrawMoneyTo” button. Then open the the Accounts dropdown. See the balance of your third Account? 101 Ether



Why is there 101 Ether and not 100.999999999some? Because we sent a transaction from Account #1 to the Smart Contract, instructing the Smart Contract to send all funds stored on the Address of the Smart Contract to the third Account in your Account-List. Gas fees were paid by Account #1. Account #3 got 1 full Ether!

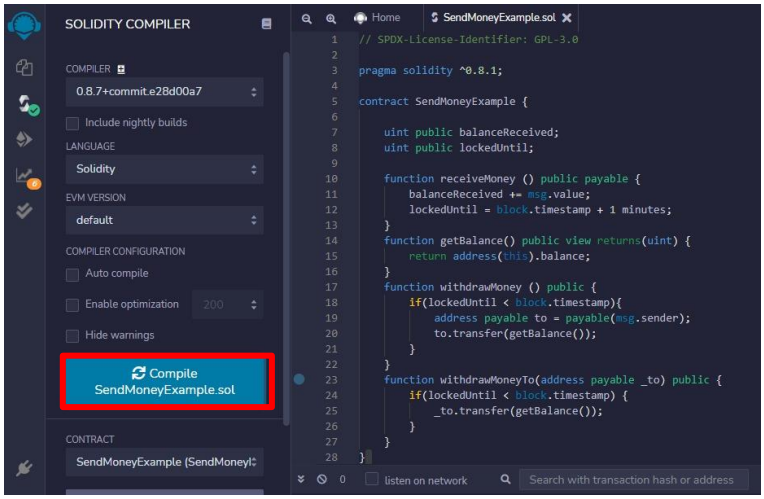
- Withdrawal Locking

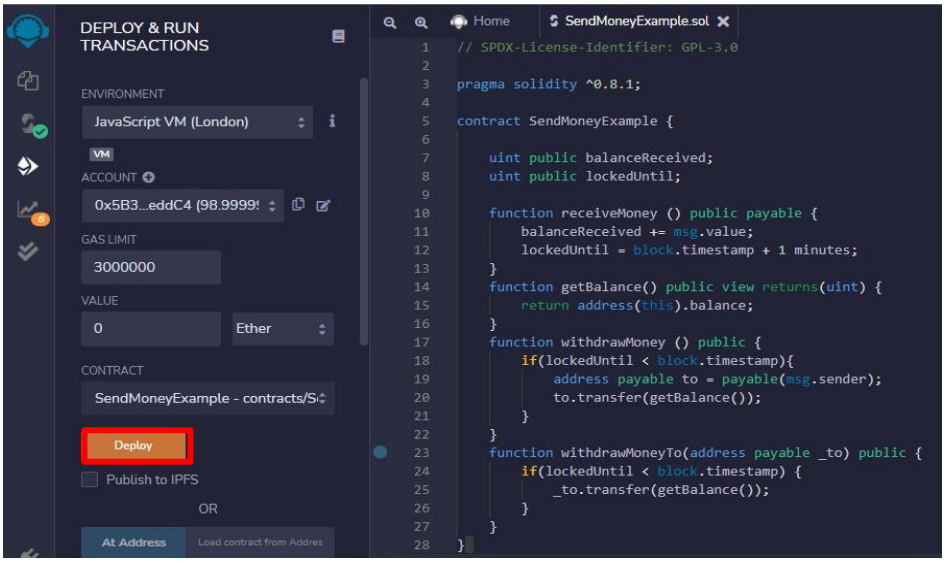
  Let's extend our Smart Contract to do some locking. What we need is to store the block.timestamp somewhere. There are several methods to go about this, I prefer to let the user know how long is it locked. So, instead of storing the deposit-timestamp, I will store the lockedUntil timestamp.



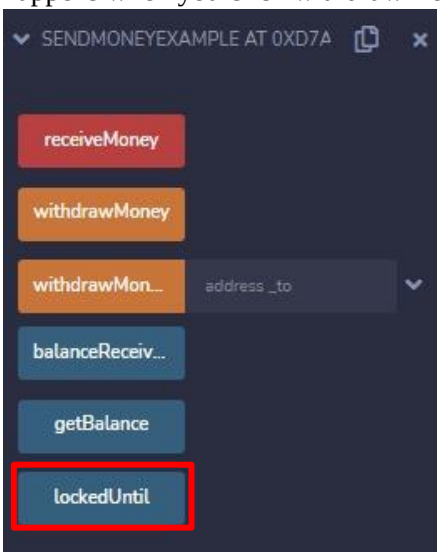
Let's deploy the Smart Contract, same procedure as before:

1. Compile and Deploy a new Instance version
2. Remove the old Instance
3. Send 1 Ether to the Smart Contract (don't forget the value field) by clicking on "receiveMoney"
4. Check the Balance!

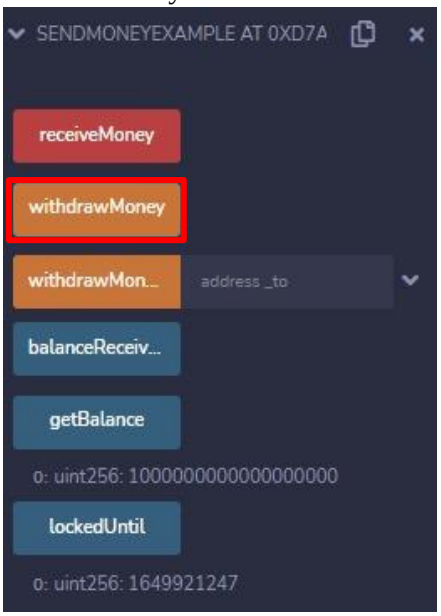




Check the "lockedUntil" by clicking on the button. It will give you the timestamp until nothing happens when you click withdrawMoney or withdrawMoneyTo .



Then, Click "withdrawMoney". The Balance stays the same until 1 Minute passed since you hit "receiveMoney".



LAB 2 : SHARED WALLET

- **Define The Smart Contract**
  This is the very basic smart contract. It can receive Ether and it's possible to withdraw Ether, but all in all, not very useful quite yet. Let's see if we can improve this a bit in the next step.

```
//SPDX-License-Identifier : MIT

pragma solidity 0.8.1;

contract Sharedwallet {
    function withdrawMoney(address payable _to. uint _amount) public {
        _to.transfer(_amount);
    }
    receive() external payable {

    }
}
```

- **Permissions : Allow only the Owner to Withdraw Ether**
  In this step we restrict withdrawal to the owner of the wallet. How can we determine the owner? It's the user who deployed the smart contract.

```
//SPDX-License-Identifier : MIT

pragma solidity 0.8.1;

contract Sharedwallet {
    address owner;
    constructor(){
        owner = msg.sender;
    }
    modifier onlyOwner(){
        require(msg.sender == owner, "You are not allowed ");
        _;
    }
    function withdrawMoney(address payable _to, uint _amount) public onlyOwner {
        _to.transfer(_amount);
    }
    receive() external payable {

    }
}
```

  Whatch out that you also add the "onlyOwner" modifier to the withdrawMoney function!

- ## Use Re-Usable Smart Contracts from OpenZeppelin

  Having the owner-logic directly in one smart contract isn't very easy to audit. Let's break it down into smaller parts and re-use existing audited smart contracts from OpenZeppelin for that. The latest OpenZeppelin contract does not have an isOwner() function anymore, so we have to create our own. Note that the owner() is a function from the Ownable.sol contract.

```
//SPDX-License-Identifier : MIT

pragma solidity 0.8.1;
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol";

contract Sharedwallet Ownable {
    function isOwner() internal view returns (bool){
        return owner() == msg.sender;
    }

    function withdrawMoney(address payable _to, uint _amount) public onlyOwner {
        _to.transfer(_amount);
    }

    receive() external payable {
    }
}
```

- ## Permissions: Add Allowances for External Roles

  In this step we are adding a mapping so we can store address => uint amounts. This will be like an array that stores [0x123546...] an address, to a specific number. So, we always know how much someone can withdraw. We also add a new modifier that checks: Is it the owner itself or just someone with allowance?

```
//SPDX-License-Identifier : MIT

pragma solidity 0.8.1;
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol";

contract Sharedwallet Ownable {
    function isOwner() internal view returns (bool){
        return owner() == msg.sender;
    }
    mapping (address => uint) public allowance;
    function addAllowance(address _who, uint _amount) public onlyOwner {
        allowance[_who] = _amount;
    }
    modifier ownerOrAllowed(uint _amount){
        require(isOwner() || allowance[msg.sender] >= _amount, "You are not allowed!");
        _;
    }
    function withdrawMoney(address payable _to, uint _amount) public ownerOrAllowed (_amount) {
        require(_amount <= adress (this).balance, "Contract doesn't own enough money");
        _to.transfer(_amount);
    }

    receive() external payable {
    }
}
```

- **Improve/Fix Allowance to avoid Double-Spending**
  Without reducing the allowance on withdrawal, someone can continuously withdraw the same amount over and over again. We have to reduce the allowance for everyone other than the owner.

```
function reduceAllowance(address _who, uint _amount) internal ownerOrAllowed(_amount) {
    allowance[_who] -= _amount;
}
modifier ownerOrAllowed(uint _amount){
    require(isOwner() || allowance[msg.sender] >= _amount, "You are not allowed!");
    _;
}
function withdrawMoney(address payable _to, uint _amount) public ownerOrAllowed (_amount) {
    require(_amount <= adress (this).balance, "Contract doesn't own enough money");
    if(!isOwner()) {
        reduceAllowance(msg.sender, _amount):
    }
    _to.transfer(_amount);
}
```

- **Improve Smart Contract Structure**
  Now we know our basic functionality, we can structure the smart contract differently. To make it easier to read, we can break the functionality down into two distinct smart contracts. Note that since Allowance is Ownable, and the SharedWallet is Allowance, therefore by commutative property, SharedWallet is also Ownable.

```
//SPDX-License-Identifier : MIT

pragma solidity 0.8.1;
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol";

contract Allowance is Ownable {
    function isOwner() internal view returns (bool){
        return owner() == msg.sender;
    }
    mapping (address => uint) public allowance;
    function setAllowance(address _who, uint _amount) public onlyOwner {
        allowance[_who] = _amount;
    }
    modifier ownerOrAllowed(uint _amount){
        require(isOwner() || allowance[msg.sender] >= _amount, "You are not allowed!");
        _;
    }
    function reduceAllowance(address _who, uint _amount) internal ownerOrAllowed(_amount) {
        allowance[_who] -= _amount;
    }
}
contract SharedWallet is Allowance {
    function withdrawMoney(address payable _to, uint _amount) public ownerOrAllowed (_amount) {
        require(_amount <= adress (this).balance, "Contract doesn't own enough money");
        if(!isOwner()) {
            reduceAllowance(msg.sender, _amount):
        }
        _to.transfer(_amount);
    }
    receive() external payable {
    }
```

Both contracts are still in the same file, so we don't have any imports (yet). That's something for another lecture later on. Right now, the important part to understand is inheritance.

- **Add Events in the Allowances Smart**

```
//SPDX-License-Identifier : MIT

pragma solidity 0.8.1;
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol";

contract Allowance is Ownable {
    event AllowanceChanged(address indexed _forWho, address indexed _byWhom, uint _oldAmount, uint _newAmount);
    mapping (address => uint) public allowance;
    function isOwner() internal view returns (bool){
        return owner() == msg.sender;
    }
    function setAllowance(address _who, uint _amount) public onlyOwner {
        emit AllowanceChanged(_who, msg.sender, allowance[_who],_amount);
        allowance[_who] = _amount;
    }
    modifier ownerOrAllowed(uint _amount){
        require(isOwner() || allowance[msg.sender] >= _amount, "You are not allowed!");
        _;
    }
    function reduceAllowance(address _who, uint _amount) internal ownerOrAllowed(_amount) {
        emit AllowanceChanged(_who, msg.sender, allowance[_who], allowance[_who] - _amount);
        allowance[_who] -= _amount;
    }
}
contract SharedWallet is Allowance {
    function withdrawMoney(address payable _to, uint _amount) public ownerOrAllowed (_amount) {
        require(_amount <= adress (this).balance, "Contract doesn't own enough money");
        if(!isOwner()) {
            reduceAllowance(msg.sender, _amount):
        }
        _to.transfer(_amount);
```

- **Add Events in the SharedWallet Smart Contract**

Obviously we also want to have events in our shared wallet, when someone deposits or withdraws funds:

```
contract SharedWallet is Allowance {
    event MoneySent(address indexed _beneficiary, uint _amount);
    event MoneyReceived(address indexed _from, uint _amount);

    function withdrawMoney(address payable _to, uint _amount) public ownerOrAllowed (_amount) {
        require(_amount <= adress (this).balance, "Contract doesn't own enough money");
        if(!isOwner()) {
            reduceAllowance(msg.sender, _amount):
        }
        emit MoneySent(_to, _amount);
        _to.transfer(_amount);
    }
    receive() external payable {
        emit MOneyReceived(msg.sender, msg.value);
    }
}
```

- **Add the SafeMath Library safeguard Mathematical Operations**

Arithmetic operations in Solidity wrap on overflow. This can easily result in bugs, because programmers usually assume that an overflow raises an error, which is the standard behavior in high level programming languages. SafeMath restores this intuition by reverting the transaction when an operation overflows. In a recent update of Solidity the Integer type variables cannot overflow anymore. You have to add safemath if you are using solidity < 0.8.

- **Remove the Renounce Ownership Functionality**
  Now, let's remove the function to remove an owner. We simply stop this with a revert. Add the following function to the SharedWallet:

```
    function renounceOwnership() public override onlyOwner {
        revert("Can't renounceOwnership here"); //not possible with this smart contract
    }
    receive() external payable {
        emit MOneyReceived(msg.sender, msg.value);
    }
}
```

- **Move the Smart Contracts into separate files**
  As a last step, let's move the smart contracts into separate files and use import functionality:
  Sharedwallet.sol

Home | Sharedwallet.sol | Allowance.sol

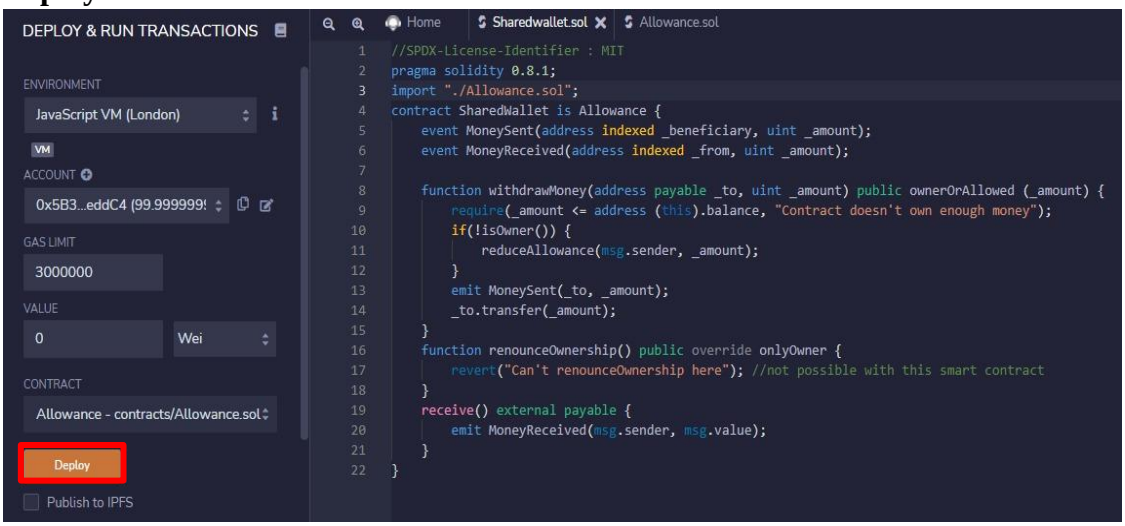
```
//SPDX-License-Identifier : MIT
pragma solidity 0.8.1;
import "./Allowance.sol";
contract SharedWallet is Allowance {
    event MoneySent(address indexed _beneficiary, uint _amount);
    event MoneyReceived(address indexed _from, uint _amount);

    function withdrawMoney(address payable _to, uint _amount) public ownerOrAllowed (_amount) {
        require(_amount <= address (this).balance, "Contract doesn't own enough money");
        if(!isOwner()) {
            reduceAllowance(msg.sender, _amount);
        }
        emit MoneySent(_to, _amount);
        _to.transfer(_amount);
    }
    function renounceOwnership() public override onlyOwner {
        revert("Can't renounceOwnership here"); //not possible with this smart contract
    }
    receive() external payable {
        emit MoneyReceived(msg.sender, msg.value);
    }
}
```

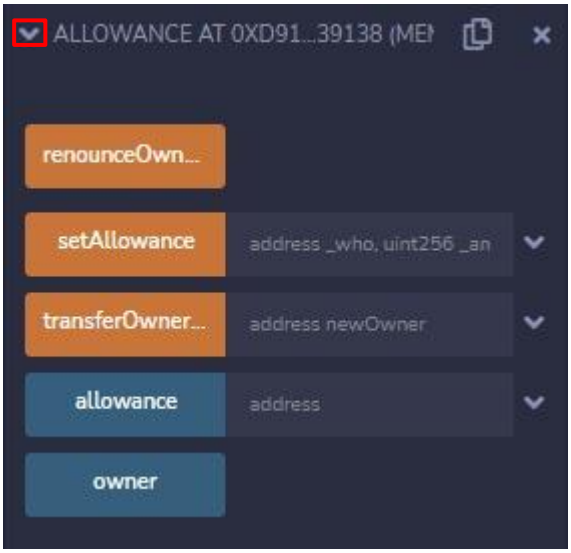
Allowance.sol

Home | Sharedwallet.sol | Allowance.sol

```
//SPDX-License-Identifier : MIT

pragma solidity 0.8.1;
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol";

contract Allowance is Ownable {
    event AllowanceChanged(address indexed _forWho, address indexed _byWhom, uint _oldAmount, uint _newAmount);
    mapping (address => uint) public allowance;
    function isOwner() internal view returns (bool){
        return owner() == msg.sender;
    }
    function setAllowance(address _who, uint _amount) public onlyOwner {
        emit AllowanceChanged(_who, msg.sender, allowance[_who],_amount);
        allowance[_who] = _amount;
    }
    modifier ownerOrAllowed(uint _amount){
        require(isOwner() || allowance[msg.sender] >= _amount, "You are not allowed!");
        _;
    }
    function reduceAllowance(address _who, uint _amount) internal ownerOrAllowed(_amount) {
        emit AllowanceChanged(_who, msg.sender, allowance[_who], allowance[_who] - _amount);
        allowance[_who] -= _amount;
    }
}
```

- **Deploy and Run the Smart Contract**





LAB 3 : SUPPLY CHAIN PROJECT

- **The ItemManager Smart Contract**

  The first thing we need is a "Management" Smart Contract. Make the ItemManager smart contract.

Home | ItemManager.sol

```
pragma solidity ^0.6.0;
contract ItemManager{
    enum SupplyChainSteps{Created, Paid, Delivered}
    struct S_Item {
        ItemManager.SupplyChainSteps _step;
        string _identifier;
        uint _priceInWei;
    }
    mapping(uint => S_Item) public items;
    uint index;
    event SupplyChainStep(uint _itemIndex, uint _step);
    function createItem(string memory _identifier, uint _priceInWei) public {
        items[index]._priceInWei = _priceInWei;
        items[index]._step = SupplyChainSteps.Created;
        items[index]._identifier = _identifier;
        emit SupplyChainStep(index, uint(items[index]._step));
        index++;
    }
    function triggerPayment(uint _index) public payable {
        require(items[_index]._priceInWei <= msg.value, "Not fully paid");
        require(items[_index]._step == SupplyChainSteps.Created, "Item is further in the supply chain");
        items[_index]._step = SupplyChainSteps.Paid;
        emit SupplyChainStep(_index, uint(items[_index]._step));
    }
    function triggerDelivery(uint _index) public {
        require(items[_index]._step == SupplyChainSteps.Paid, "Item is further in the supply chain");
        items[_index]._step = SupplyChainSteps.Delivered;
        emit SupplyChainStep(_index, uint(items[_index]._step));
    }
}
```

With this smart contract, it's possible to add items and pay them, move them forward in the supply chain and trigger a delivery.

- **Item Smart Contract**

  Make another smart contract and give name Item.

Home | Item.sol

```
pragma solidity ^0.6.0;

import "./ItemManager.sol";
contract Item {
    uint public priceInWei;
    uint public paidWei;
    uint public index;

    ItemManager parentContract;
    constructor(ItemManager _parentContract, uint _priceInWei, uint _index) public {
        priceInWei = _priceInWei;
        index = _index;
        parentContract = _parentContract;
    }
    receive() external payable {
        require(msg.value == priceInWei, "We don't support partial payments");
        require(paidWei == 0, "Item is already paid!");
        paidWei += msg.value;
        (bool success, ) = address(parentContract).call{value:msg.value}(abi.encodeWithSignature("triggerPayment(uint256)", index));
        require(success, "Delivery did not work");
    }
    fallback () external {
    }
}
```

Change the ItemManager Smart Contract to use the Item Smart Contract instead of the Struct only:

```
pragma solidity ^0.6.0;
import "./Item.sol";
contract ItemManager{
    enum SupplyChainSteps{Created, Paid, Delivered}
    struct S_Item {
        ItemManager.SupplyChainSteps _step;
        string _identifier;
        uint _priceInWei;
    }
    mapping(uint => S_Item) public items;
    uint index;
    enum SupplyChainSteps {Created, Paid, Delivered}
    event SupplyChainStep(uint _itemIndex, uint _step, address _address);
    function createItem(string memory _identifier, uint _priceInWei) public {
        Item item = new Item(this, _priceInWei, index);
        items[index]._item = item;
        items[index]._step = SupplyChainSteps.Created;
        items[index]._identifier = _identifier;
        emit SupplyChainStep(index, uint(items[index]._step), address(item));
        index++;
    }
    function triggerPayment(uint _index) public payable {
        Item item = items[_index]._item;
        require(address(item) == msg.sender, "Only items are allowed to update themselves");
        require(item.priceInWei() == msg.value, "Not fully paid yet");
        require(items[_index]._step == SupplyChainSteps.Created, "Item is further in the supply chain");
        items[_index]._step = SupplyChainSteps.Paid;
        emit SupplyChainStep(_index, uint(items[_index]._step));
    }
    function triggerDelivery(uint _index) public {
        require(items[_index]._step == SupplyChainSteps.Paid, "Item is further in the supply chain");
        items[_index]._step = SupplyChainSteps.Delivered;
        emit SupplyhaCinStep(_index, uint(items[_index]._step), address(items_index]._item));
    }
}
```

Now with this we just have to give a customer the address of the Item Smart Contract created during "createItem" and he will be able to pay directly by sending X Wei to the Smart Contract.

- **Ownable Functionality**

Home | Item.sol | ItemManager.sol | Ownable.sol

```
pragma solidity ^0.6.0;

contract Ownable {
    address public _owner;

    constructor () internal{
        _owner = msg.sender;
    }
    /*@dev Throws if called by any account other than the owner.*/
    modifier onlyOwner(){
        require(isOwner(), "Ownable: caller is not the owner");
        _;
    }
    /*@dev Returns true if the caller is the current owner.*/
    function isOwner() public view returns (bool) {
        return (msg.sender == _owner);
    }
}
```

Then modify the ItemManager, so that all function should be executable by the “owner only”.

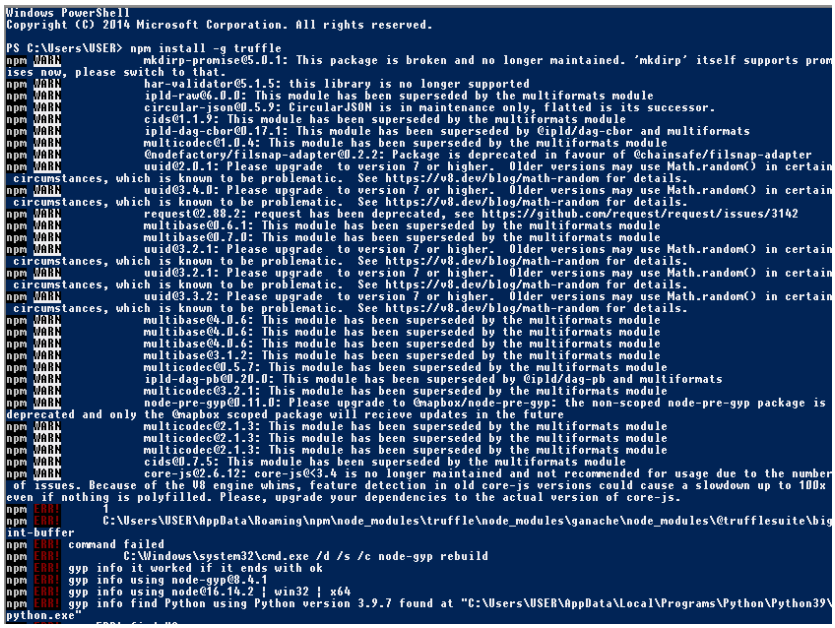
```
pragma solidity ^0.6.0;
import "./Ownable.sol"
import "./Item.sol";
contract ItemManager is Ownable {
    enum SupplyChainSteps{Created, Paid, Delivered}
    struct S_Item {
        ItemManager.SupplyChainSteps _step;
        string _identifier;
        uint _priceInWei;
    }
    mapping(uint => S_Item) public items;
    uint index;
    enum SupplyChainSteps {Created, Paid, Delivered}
    event SupplyChainStep(uint _itemIndex, uint _step, address _address);
    function createItem(string memory _identifier, uint _priceInWei) public onlyOwner {
        Item item = new Item(this, _priceInWei, index);
        items[index]._item = item;
        items[index]._step = SupplyChainSteps.Created;
        items[index]._identifier = _identifier;
        emit SupplyChainStep(index, uint(items[index]._step), address(item));
        index++;
    }
    function triggerPayment(uint _index) public payable {
        Item item = items[_index]._item;
        require(address(item) == msg.sender, "Only items are allowed to update themselves");
        require(item.priceInWei() == msg.value, "Not fully paid yet");
        require(items[_index]._step == SupplyChainSteps.Created, "Item is further in the supply chain");
        items[_index]._step = SupplyChainSteps.Paid;
        emit SupplyChainStep(_index, uint(items[_index]._step));
    }
    function triggerDelivery(uint _index) public onlyOwner {
        require(items[_index]._step == SupplyChainSteps.Paid, "Item is further in the supply chain");
        items[_index]._step = SupplyChainSteps.Delivered;
        emit SupplyhaCinStep(_index, uint(items[_index]._step), address(items_index]._item));
    }
```

- **Install Truffle**

To install truffle open a Powershell for Windows
Type in :

```
npm install –g truffle
```

When i am trying to install the truffle in my PC, i’ve got some error on my Windows PowerShell.



Then, i continue this documentation using Windows PowerShell picture from the Ethereum Blockchain Developer Guide.

Install the truffle in Windows PowerShell

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

thoma> npm install -g truffle
C:\Users\thoma\AppData\Roaming\npm\truffle -> C:\Users\thoma\AppDat

> truffle@5.1.8 postinstall C:\Users\thoma\AppData\Roaming\npm\node
> node ./scripts/postinstall.js

- Fetching solc version list from solc-bin. Attempt #1
+ truffle@5.1.8
updated 1 package in 9.586s
thoma>
```

Then create an empty folder, in this case I am creating "s06-eventtrigger"
Type in:

```
mkdir s06-eventtrigger
cd s06-eventtrigger
ls
```

```
ebd> mkdir s06-eventtrigger


    Directory: C:\101Tmp\ebd


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        1/11/2020  10:39 AM                s06-eventtrigger


ebd> cd .\s06-eventtrigger\
s06-eventtrigger> ls
s06-eventtrigger>
```

And unbox the react box:
Type in:

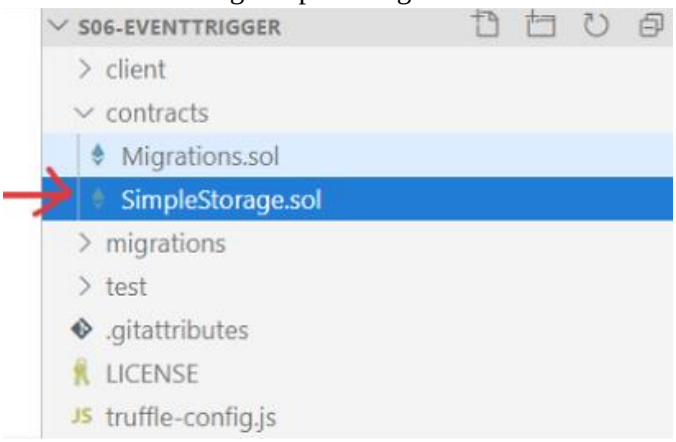
```
Truffle unbox react
```

This should download a repository and install all dependencies in the current folder:

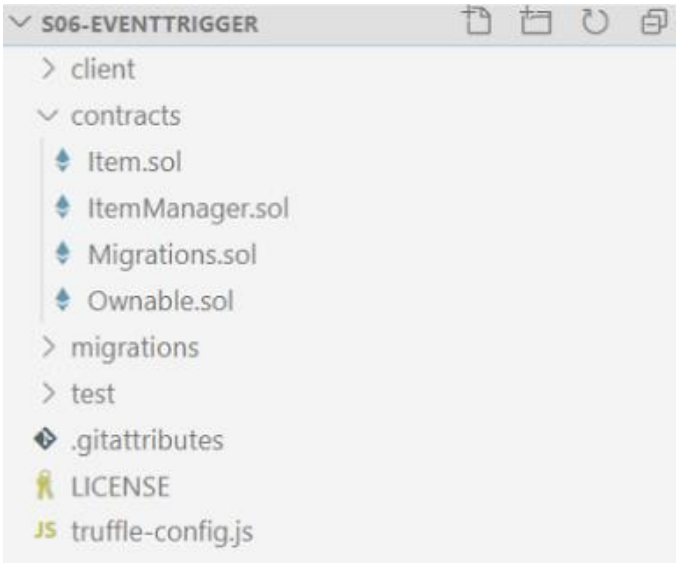
```
s06-eventtrigger> truffle unbox react
√ Preparing to download box
√ Downloading
√ cleaning up temporary files
√ Setting up box
s06-eventtrigger> ls


    Directory: C:\101Tmp\ebd\s06-eventtrigger


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        1/11/2020  10:41 AM                client
d-----        1/11/2020  10:41 AM                contracts
d-----        1/11/2020  10:41 AM                migrations
d-----        1/11/2020  10:41 AM                test
-a----        1/11/2020  10:41 AM             33 .gitattributes
-a----        1/11/2020  10:41 AM           1075 LICENSE
-a----        1/11/2020  10:41 AM            297 truffle-config.js


s06-eventtrigger>
```

- **Add Contracts**

Remove the existing SimpleStorage Smart Contract but leave the "Migrations.sol" file:



Add in our Files:



Then modify the "migration" file in the migrations/ folder:

```
const path = require("path");

module.exports = {
    // See <http://truffleframework.com/docs/advanced/configuration>
    // to customize your Truffle configuration!
    contracts_build_directory: path.join(__dirname, "client/src/contracts"),
    networks: {
        develop: {
            port: 8545
            }
    },
    compilers: {
        solc: {
            version: "^0.6.0"
            }
    }
};
```

Run the truffle develop console to check if everything is alright and can be migrated. On the Windows PowerShell run.

```
truffle develop
```

and then simply type in :

```
migrate
```

```
truffle(develop)> migrate

Compiling your contracts...
===========================
> Compiling .\contracts\Item.sol
> Compiling .\contracts\ItemManager.sol
> Compiling .\contracts\Ownable.sol
> Artifacts written to C:\101Tmp\ebd\s06-eventtrigger\client\src\
> Compiled successfully using:
   - solc: 0.6.1+commit.e6f7d5a4.Emscripten.clang



Starting migrations...
======================
```

- **Modify HTML**

  Now it's time that we modify our HTML so we can actually interact with the Smart Contract from the Browser. Open "client/App.js" and modify a few things inside the file:

```
import React, { Component } from "react";
import ItemManager from "./contracts/ItemManager.json";
import Item from "./contracts/Item.json";
import getWeb3 from "./getWeb3";
import "./App.css";
class App extends Component {
    state = {cost: 0, itemName: "exampleItem1", loaded:false};
    componentDidMount = async () => {
        try {
            // Get network provider and web3 instance.
            this.web3 = await getWeb3();
            // Use web3 to get the user's accounts.
            this.accounts = await this.web3.eth.getAccounts();
            // Get the contract instance.
            const networkId = await this.web3.eth.net.getId();
            this.itemManager = new this.web3.eth.Contract(
                ItemManager.abi,
                ItemManager.networks[networkId] && ItemManager.networks[networkId].address,
            );
            this.item = new this.web3.eth.Contract(
                Item.abi,
                Item.networks[networkId] && Item.networks[networkId].address,
            );
            this.setState({loaded:true});
        } catch (error) {
            // Catch any errors for any of the above operations.
            alert(
                `Failed to load web3, accounts, or contract. Check console for details.`,
            );
            console.error(error);
        }
    };
//.. more code here ...
```

Then add in a form to the HTML part on the lower end of the App.js file, in the "render" function:

```
render() {
    if (!this.state.loaded) {
        return <div>Loading Web3, accounts, and contract...</div>;
        }
        return (
            <div className="App">
            <h1>Simply Payment/Supply Chain Example!</h1>
            <h2>Items</h2>
            <h2>Add Element</h2>
            Cost: <input type="text" name="cost" value={this.state.cost} onChange={this.handleInputChange} />
            Item Name: <input type="text" name="itemName" value={this.state.itemName} onChange={this.handleInputChange} />
            <button type="button" onClick={this.handleSubmit}>Create new Item</button>
        </div>
    );
}
```

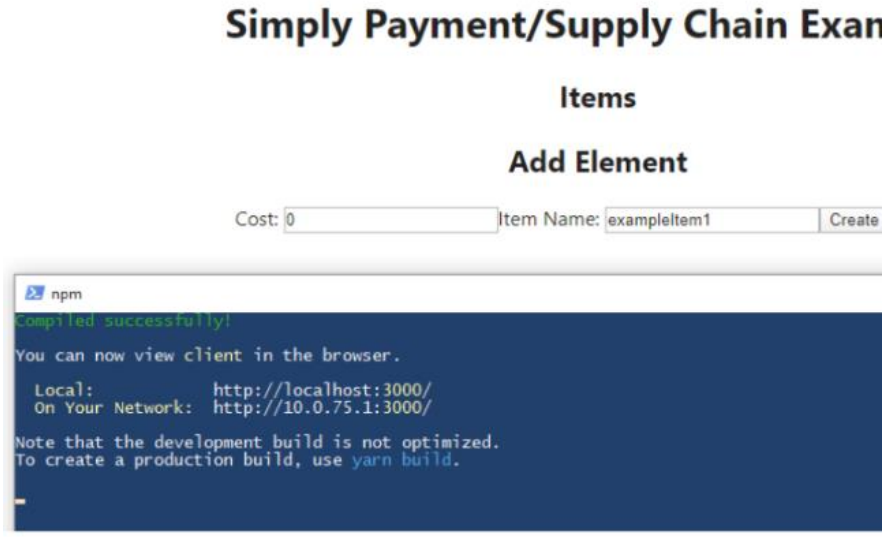
And add two functions, one for handleInputChange, so that all input variables are set correctly. And one for sending the actual transaction off to the network:

```
handleSubmit = async () => {
    const { cost, itemName } = this.state;
    console.log(itemName, cost, this.itemManager);
    let result = await this.itemManager.methods.createItem(itemName, cost).send({ from: this.accounts[0] });
    console.log(result);
    alert("Send "+cost+" Wei to "+result.events.SupplyChainStep.returnValues._address);
   };
  handleInputChange = (event) => {
      const target = event.target;
      const value = target.type === 'checkbox' ? target.checked : target.value;
      const name = target.name;
      this.setState({
          [name]: value
      });
}
```

Open another terminal/powershell (leave the one running that you have already opened with truffle) and go to the client folder and run.
Type in :

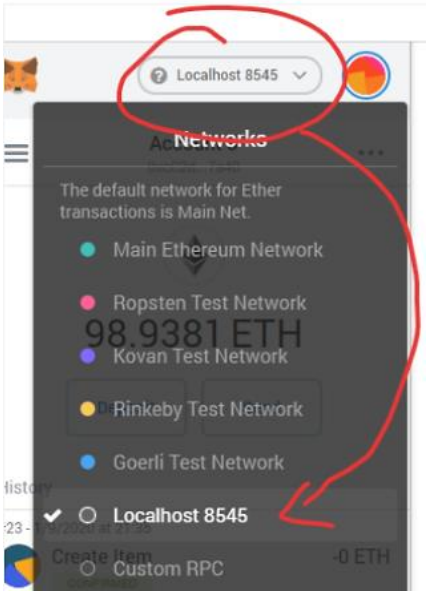
```
npm start
```

This will start the development server on port 3000 and should open a new tab in your browser:



- **Connect with MetaMask**
  In this section we want to connect our React App with MetaMask and use MetaMask as a Keystore to sign transactions. It will also be a proxy to the correct blockchain.

  First, connect with MetaMask to the right network:

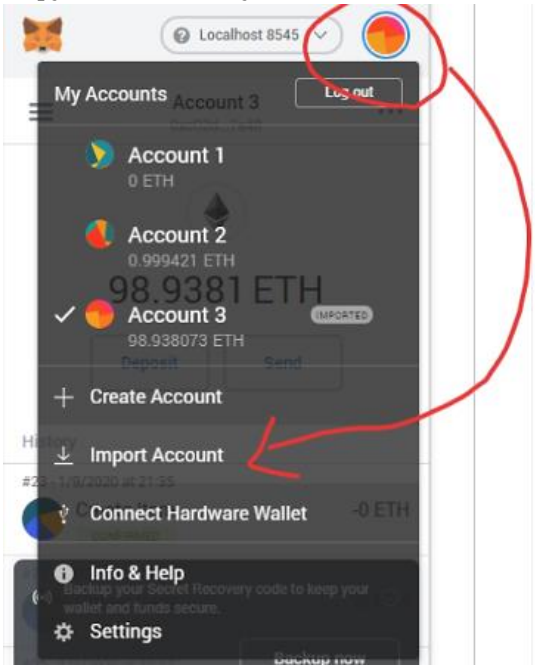


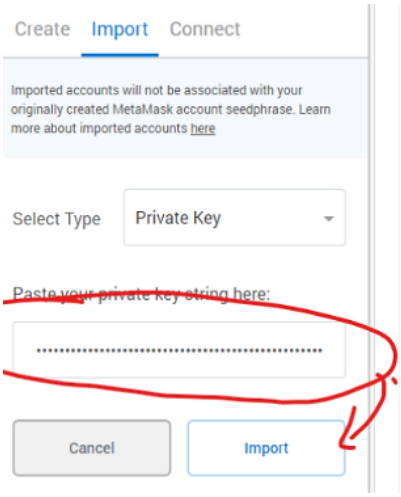
When we migrate the smart contracts with Truffle Developer console, then the first account in the truffle developer console is the "owner". So, either we disable MetaMask in the Browser to interact with the app or we add in the private key from truffle developer console to MetaMask.

In the Terminal/Powershell where Truffle Developer Console is running scroll to the private keys on top:

```
Private Keys:
(0) 2a9ed36cdb66f81093a82443c2b9f237f3534ef75f4f044fa6ebd76d5d05f61
(1) f9c941a67e63fe4b84fe63ad652c29b2f225eb57562b246bf44bd3527b94b48
```

Copy the Private Key and add it into MetaMask:





Then your new Account should appear here with ~100 Ether in it.
Now let's add a new Item to our Smart Contract. You should be presented with the popup to send the message to an end-user.

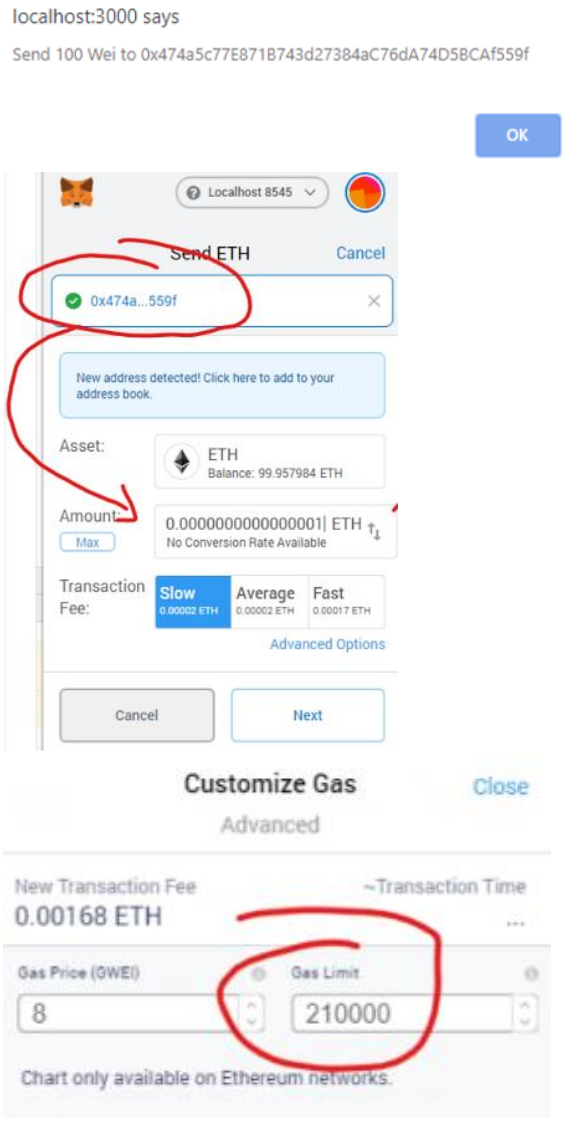
- **Listen to Payments**
  There are multiple ways to solve this particular issue. For example you could poll the Item smart contract. You could watch the address on a low-level for incoming payments. But that's not what we want to do. What we want is to wait for the event "SupplyChainStep" to trigger with _step == 1 (Paid).
  Let's add another function to the App.js file:

```
listenToPaymentEvent = () => {
    let self = this;
    this.itemManager.events.SupplyChainStep().on("data", async function(evt) {
        if(evt.returnValues._step == 1) {
            let item = await self.itemManager.methods.items(evt.returnValues._itemIndex).call();
            console.log(item);
            alert("Item " + item._identifier + " was paid, deliver it now!");
        };
        console.log(evt);
    });
}
```

And call this function when we initialize the app in "componentDidMount":

```
//...
this.item = new this.web3.eth.Contract(
    ItemContract.abi,
    ItemContract.networks[this.networkId] && ItemContract.networks[this.networkId].address,
    );
    // Set web3, accounts, and contract to the state, and then proceed with an
    // example of interacting with the contract's methods.
    this.listenToPaymentEvent();
    this.setState({ loaded:true });
    } catch (error) {
      // Catch any errors for any of the above operations.
        alert(
          `Failed to load web3, accounts, or contract. Check console for details.`,
        );
        console.error(error);
    }
//...
```

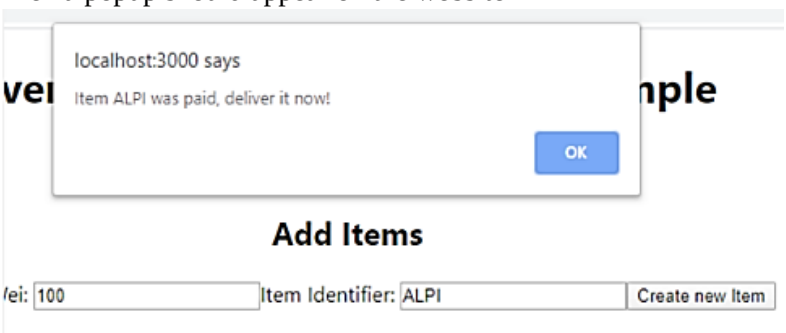
Whenever someone pays the item a new popup will appear telling you to deliver. You could also add this to a separate page, but for simplicity we just add it as an alert popup to showcase the trigger-functionality:



Take the address, give it to someone telling them to send 100 wei (0.0000000000000001 Ether) and a bit more gas to the specified address. You can do this either via MetaMask or via the truffle console:

```
web3.eth.sendTransaction({to: "ITEM_ADDRESS", value: 100, from: accounts[1], gas: 2000000});
```

Then a popup should appear on the website



- **Unit Test**

  There is something special in Truffle about unit testing. The problem is that in the testing suite you get contract-abstractions using truffle-contract, while in the normal app you worked with web3-contract instances.

  Let's implement a super simple unit test and see if we can test that items get created.

```
const ItemManager = artifacts.require("./ItemManager.sol");

contract("ItemManager", accounts => {
it("... should let you create new Items.", async () => {
    const itemManagerInstance = await ItemManager.deployed();
    const itemName = "test1";
    const itemPrice = 500;

    const result = await itemManagerInstance.createItem(itemName, itemPrice, { from: accounts[0] });
    assert.equal(result.logs[0].args._itemIndex, 0, "There should be one item index in there")
    const item = await itemManagerInstance.items(0);
    assert.equal(item._identifier, itemName, "The item has a different identifier");
});
});
```

  Keep the truffle development console open and type in a new PowerShell window:

```
truffle test
```

  It should bring up a test like this:

```
Compiling your contracts...
===========================
> Compiling .\contracts\Item.sol
> Compiling .\contracts\ItemManager.sol
> Compiling .\contracts\Ownable.sol


  Contract: ItemManager
    √ ... should let you create new Items. (160ms)


  1 passing (216ms)

s06-eventtrigger>
```

  This is how you add unit tests to your smart contracts.