

📁 Solidity, Truffle & React Project 📁

Event-Triggers / Supply Chain Example

Copyright © 2019-2020 Thomas Wiesner – <https://vomtom.at>

The included Source-Code shall be Distributed under the License: "[Attribution CC BY 4.0](#)"

Real-World Use-Case for this Project

- Can be part of a supply-chain solution
- Automated Dispatch upon payment
- Payment collection without middlemen

Development-Goal

- Showcase Event-Triggers
- Understand the low-level function `address.call.value()()`
- Understand the Workflow with Truffle
- Understand Unit Testing with Truffle
- Understand Events in HTML

Contents

Step 1 – The ItemManager Smart Contract	2
Step 2 – The Item Smart Contract	2
Step 3 – Add OnlyOwner Modifiers and Ownable Functionality	5
Step 4 – Install Truffle and Unbox your First Project	6
Step 5 – Add in our Contracts	8
Step 6 – Modify HTML to Add Items to the ItemManager	10
Step 7 – Connect with MetaMask and add Private Key to MetaMask	13
Step 8 – Listen to Payments	15
Step 9 – Unit Test the functionality	17

Step 1 – The ItemManager Smart Contract

The first thing we need is a “Management” Smart Contract, where we can add items.

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

But that's something I don't like, because ideally I just want to give the user a simple address to send money to.

Step 2 – The Item Smart Contract

Let's add another smart contract:

```
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 {

    }
}
```

And 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 {

    struct S_Item {
        Item _item;
        ItemManager.SupplyChainSteps _step;
        string _identifier;
    }
    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), address(item));
    }

    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), 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. But the smart contract isn’t very secure yet. We need some sort of owner functionality.

Step 3 – Add OnlyOwner Modifiers and Ownable Functionality

Normally we would add the OpenZeppelin Smart Contracts with the Ownable Functionality. But at the time of writing this document they are not updated to solidity 0.6 yet. So, instead we will add our own Ownable functionality very much like the one from OpenZeppelin:

```
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);
    }
}
```

```
pragma solidity ^0.6.0;

import "./Ownable.sol";
import "./Item.sol";

contract ItemManager is Ownable {

    //...

    function createItem(string memory _identifier, uint _priceInWei) public onlyOwner {
    //...
    }

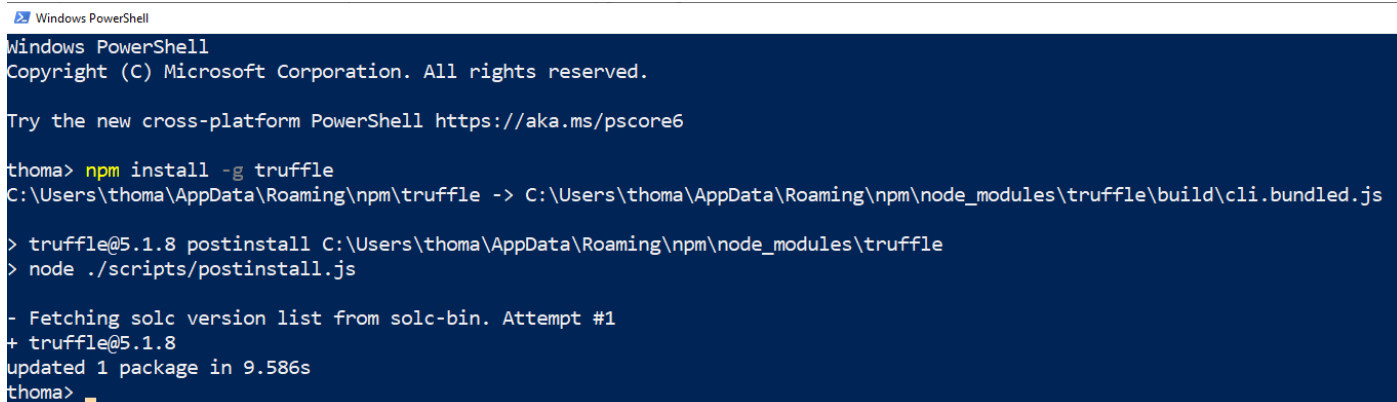
    function triggerPayment(uint _index) public payable {
    //...
    }

    function triggerDelivery(uint _index) public onlyOwner {
    //...
    }
}
```

Step 4 – Install Truffle and Unbox your First Project

To install truffle open a terminal (Mac/Linux) or a PowerShell (Windows 10)

Type in: `npm install -g truffle`



```
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\AppData\Roaming\npm\node_modules\truffle\build\cli.bundled.js

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

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

Hint: I am working here with version 5.1.8 of Truffle. If you want to follow the exact same version then type in `npm install -g truffle@5.1.8`

Then create an empty folder, in this case I am creating “s06-eventtrigger”

```

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:

```
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

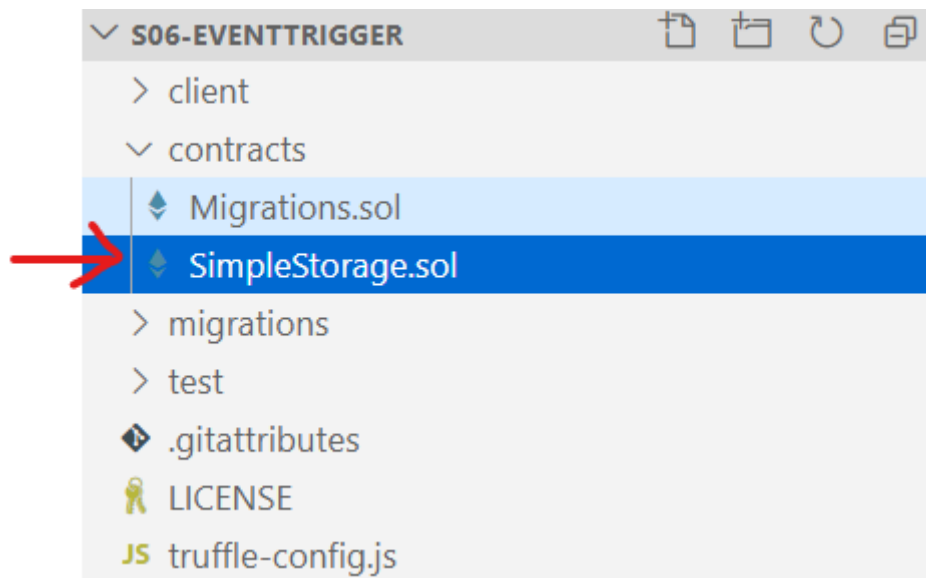
Mode                LastWriteTime         Length Name
----                -
d-----          1/11/2020  10:41 AM                client
d-----          1/11/2020  10:41 AM                contracts
d-----          1/11/2020  10:41 AM                migrations
d-----          1/11/2020  10:41 AM                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>

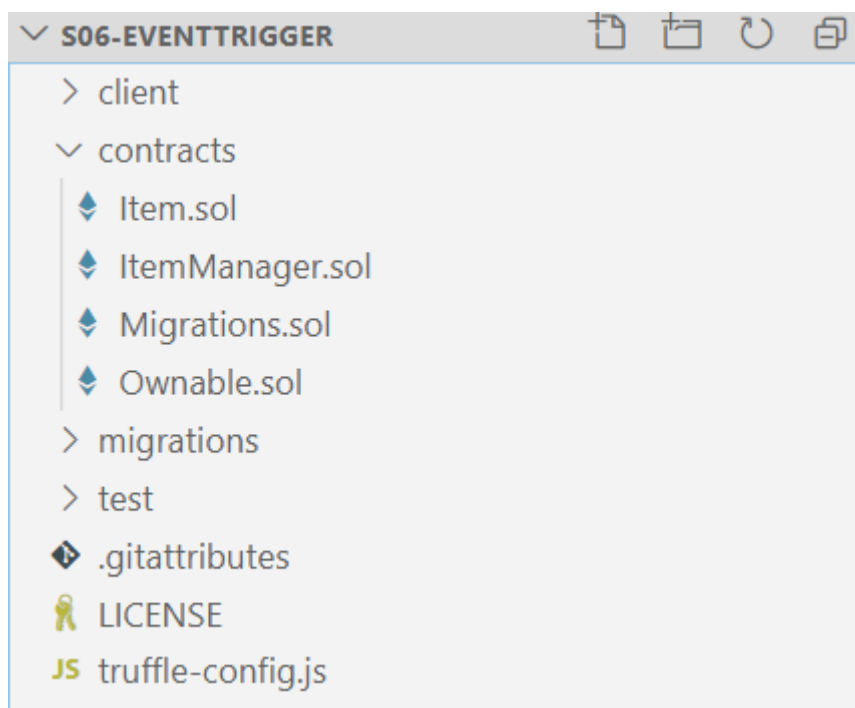
```

Step 5 – Add in our Contracts

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



Add in our Files:



Then modify the “migration” file:

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

module.exports = function(deployer) {
  deployer.deploy(ItemManager);
};
```

Modify the truffle-config.js file to lock in a specific compiler version:


```
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: "0.6.1"
    }
  }
};
```

Run the truffle develop console to check if everything is alright and can be migrated:

On the terminal/powershell run

```
truffle develop
```

and then

```
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\contracts
> Compiled successfully using:
   - solc: 0.6.1+commit.e6f7d5a4.Emscripten.clang

Starting migrations...
=====
> Network name:      'develop'
```

Step 6 – Modify HTML to Add Items to the ItemManager

Open “client/App.js” and modify the beginning of 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 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.handle
InputChange} />
      Item Name: <input type="text" name="itemName" value={this.state.itemName} onChange
={this.handleChange} />
      <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) and go to the client folder and run

```
npm start
```

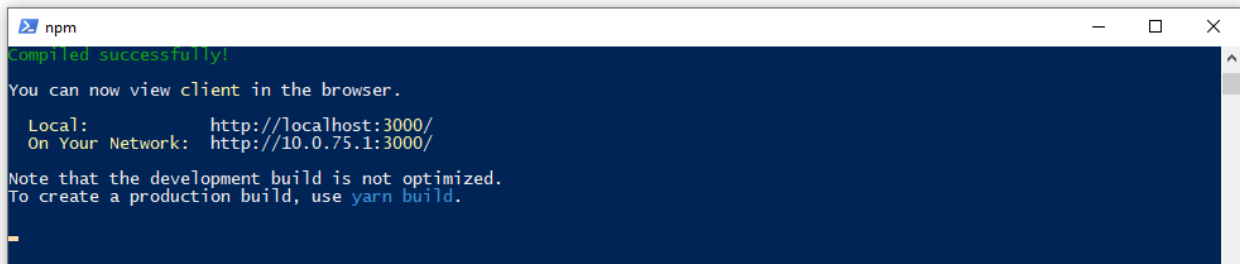
This will start the development server on port 3000 and should open a new tab in your browser:

Simply Payment/Supply Chain Example!

Items

Add Element

Cost: Item Name:

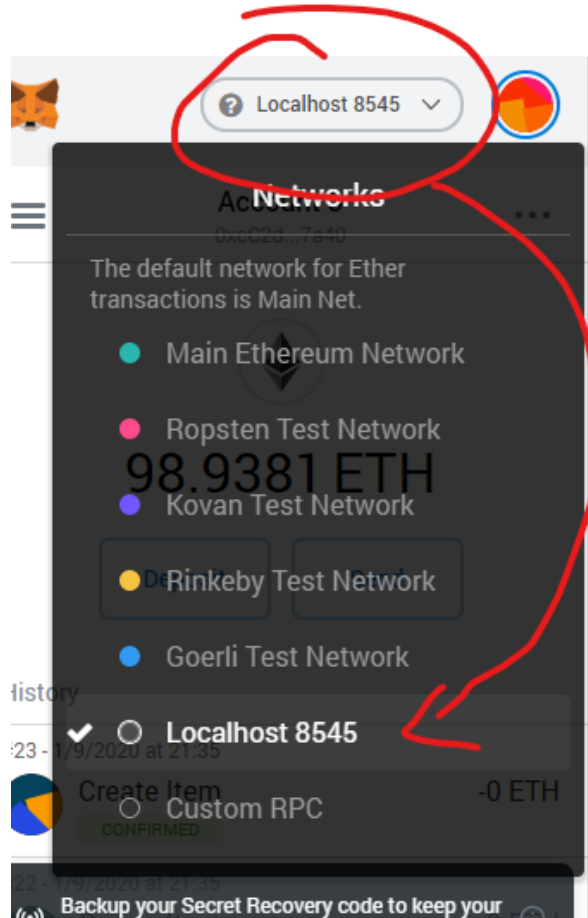


```
npm
Compiled successfully!
You can now view client in the browser.
  Local:      http://localhost:3000/
  On Your Network: http://10.0.75.1:3000/
Note that the development build is not optimized.
To create a production build, use yarn build.
```

If you see an error message that the network wasn't found or the contract wasn't found under the address provided – don't worry: Follow along in the next step where you change the network in MetaMask! As long as there is no error in your terminal and it says **"Compiled successfully"** you're good to go!

Step 7 – Connect with MetaMask and add Private Key to MetaMask

First, connect with MetaMask to the right network:



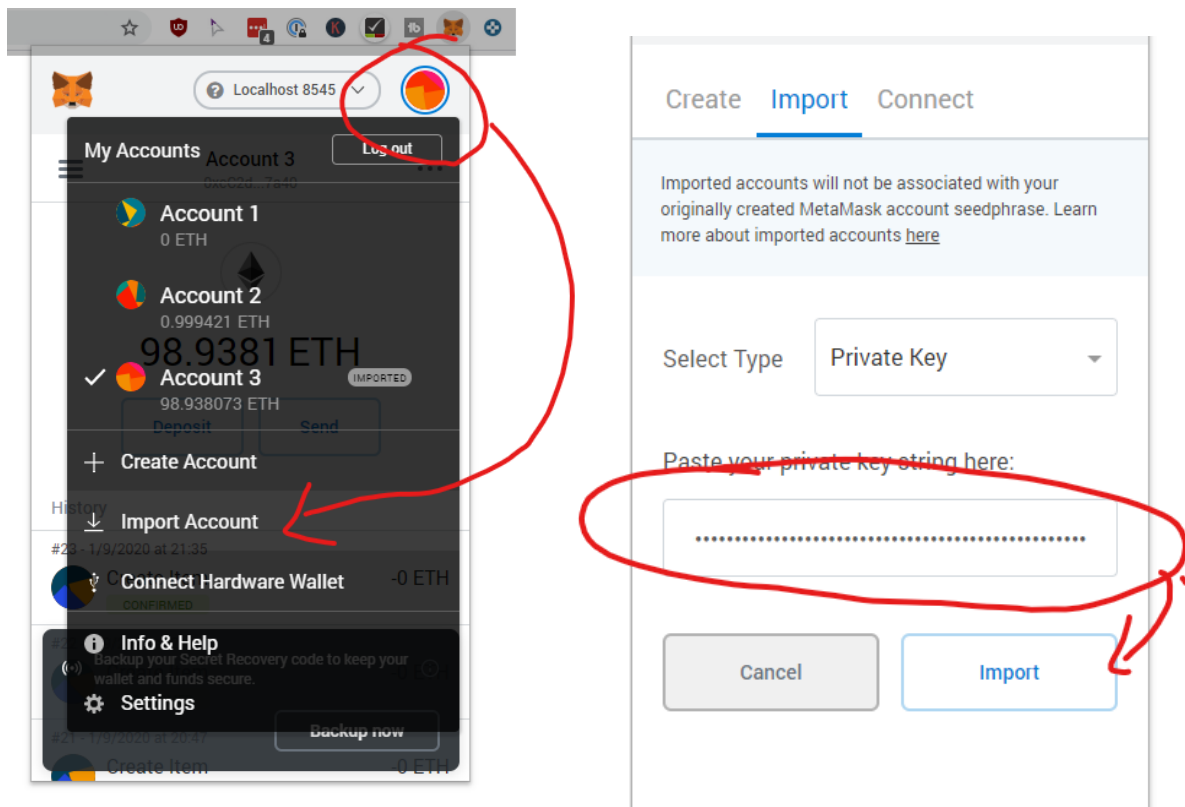
When we migrate the smart contracts with Truffle Developer console, then the first account in the truffle developer console is the “owner”. So, either we disable MetaMask in the Browser to interact with the app or we add in the private key from truffle developer console to MetaMask.

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

Private Keys:

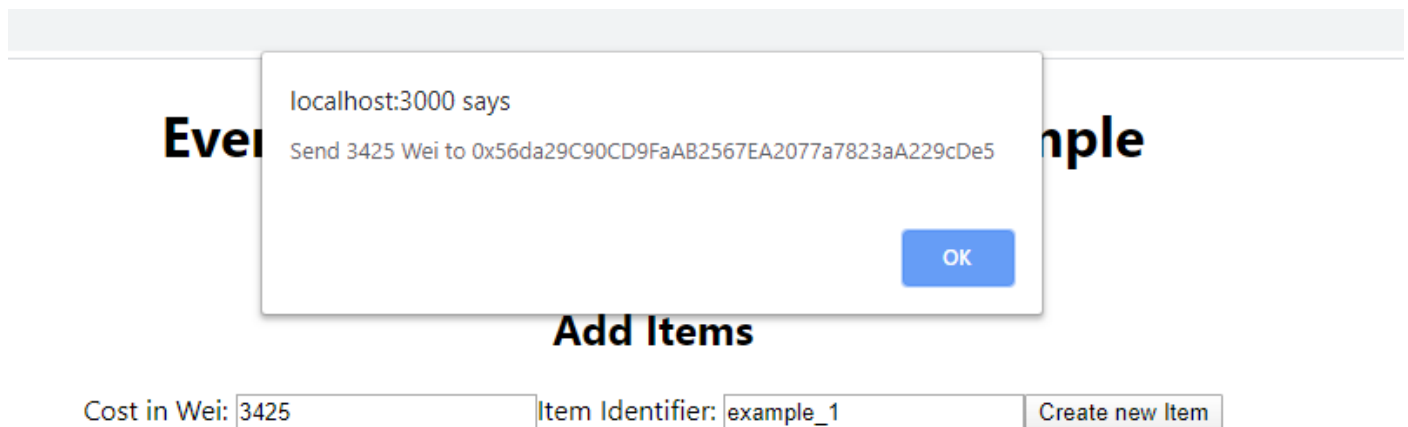
```
(0) 2a9ed36cdb66f81093a82443c2b9f237f3534ef75f4f044fa6ebd76d5d05f615
(1) f9c941a67e63fe4b84fe63ad652c29b2f225eb57562b246bf44bd3527b94b486
```

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.



Step 8 – Listen to Payments

Now that we know *how much* to pay to *which address* we need some sort of feedback. Obviously we don't want to wait until the customer tells us he paid, we want to know right on the spot if a payment happened.

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:

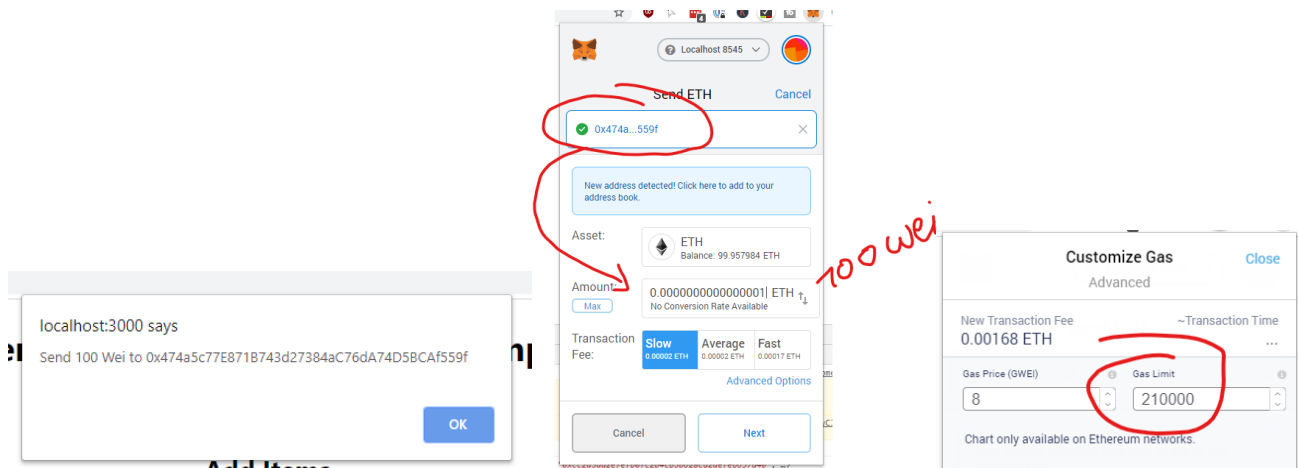
```
listenToPaymentEvent = () => {
  let self = this;
  this.itemManager.events.SupplyChainStep().on("data", async function(evt) {
    if(evt.returnValues._step == 1) {
      let item = await self.itemManager.methods.items(evt.returnValues._itemIndex).call(
    );
      console.log(item);
      alert("Item " + 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].ad
dress,
);

// 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);
}
//...
```

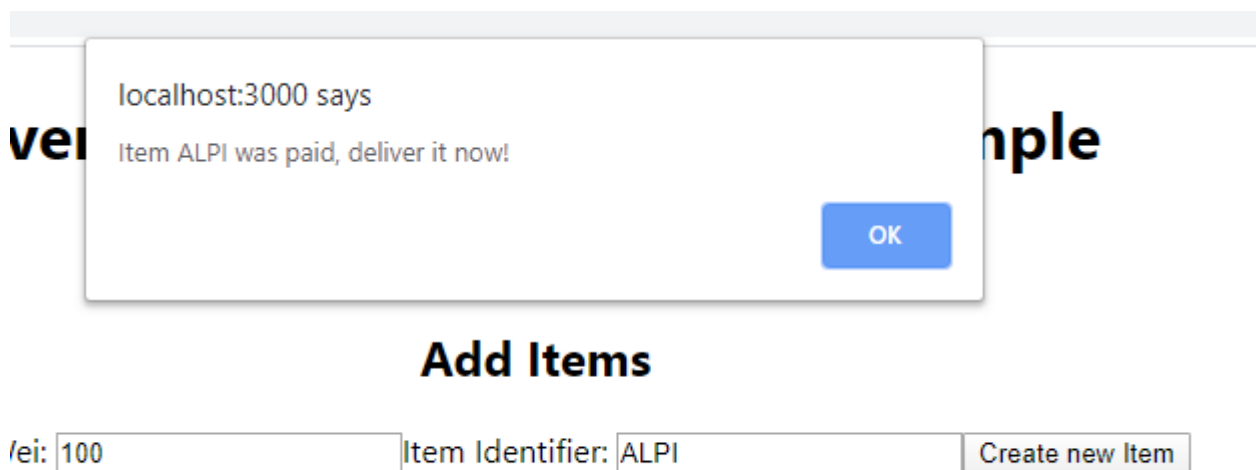
Whenever someone pays the item a new popup will appear telling you to deliver. You could also add this to a separate page, but for simplicity we just add it as an alert popup to showcase the trigger-functionality:



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



Step 9 – Unit Test the functionality

Unit testing is important, that's out of the question. But how to write unit tests?

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.

First of all, delete the tests in the "/test" folder. They are for the simplestorage smart contract which doesn't exist anymore. Then add new tests:

```
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");
  });
});
```

Mind the difference: In web3js you work with "instance.methods.createItem" while in truffle-contract you work with "instance.createItem". Also, the events are different. In web3js you work with result.events.returnValues and in truffle-contract you work with result.logs.args. The reason is that truffle-contract mostly took the API from web3js 0.20 and they did a major refactor for web3js 1.0.0.

Keep the truffle development console open and type in a new terminal/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.

Congratulations, LAB is completed



From the Course “Ethereum Blockchain Developer – Build Projects in Solidity”



FULL COURSE:

<https://www.udemy.com/course/blockchain-developer/?referralCode=E8611DF99D7E491DFD96>