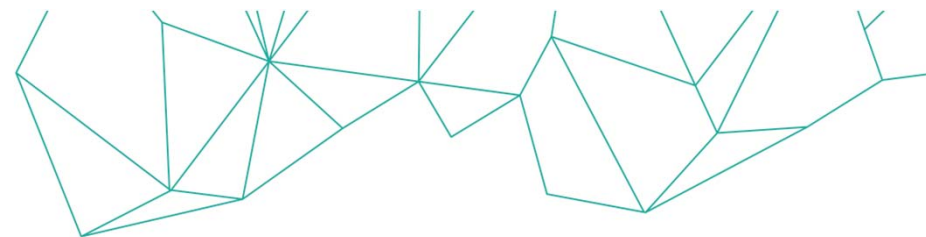




DECENTRALIZED
— TRAINING SERIES



Hands-on Session Case-study

A blockchain dApp for the Supply Chain and Logistics Industry

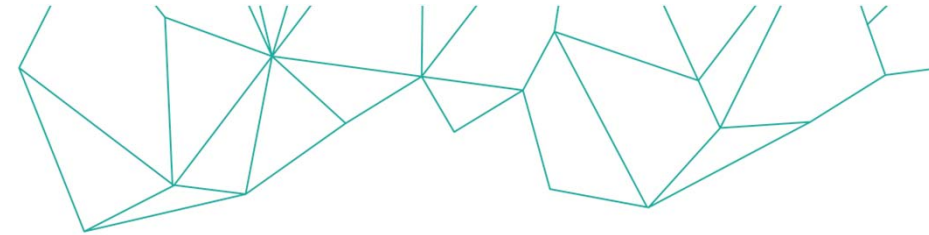
Dr. Christos Filelis – Papadopoulos

Information Technologies Institute (ITI), Centre for Research and Technologies Hellas (CERTH)

Dr. Klitos Christodoulou

University of Nicosia (UNIC)

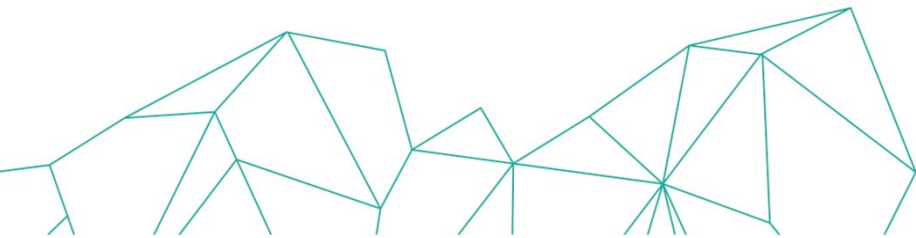




Decentralized Training Series Workshop: Certified Ethereum Specialist

19-20 November 2018, Athens, Greece

I / Ethereum Blockchain (Back End)

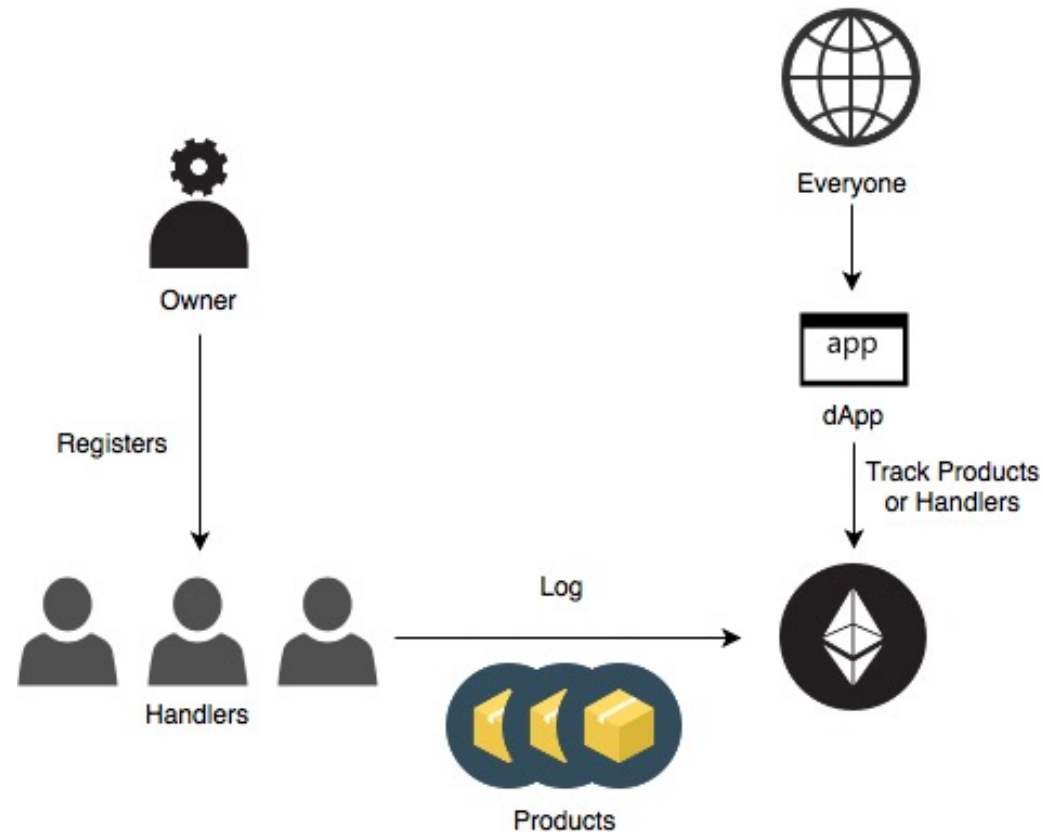


Basic Requirements for the Distributed Application

▼ Supply Chain and Logistics requirements:

- ▼ The app should focus on tracking of products as they traverse the supply chain and are handled by different handlers.
- ▼ The Owner of the contract will be responsible for registering handlers.
- ▼ Each registered handler is responsible for logging the creation or handling of products. Thus, the path of the product is easily traceable and verifiable by everyone on the Ethereum network, through a dApp or a an explorer such as Etherscan.
- ▼ The contract should return data concerning the handlers, products and checkpoints per product.
- ▼ The dApp is focused on promoting transparency in the movement of products and goods.

A simple schematic representation



Flow of the application

- ▼ The **Owner** adds **Handlers**.
- ▼ **Handlers** add **Products**.
- ▼ **Handlers** add **Checkpoints** to **Products**.
- ▼ Retrieval of information can be performed by every **Ethereum** user:
 - ▼ Retrieve info of a **Handler** based on Ethereum address
 - ▼ Retrieve all **Handlers** and their info
 - ▼ Retrieve info of a **Product** based on its Unique ID, along (with info for all Checkpoints)
 - ▼ Retrieve info of all **Products** (without info for Checkpoints)

Properties of a Handler

▼ Each **Handler** is identified by:

- ▼ An Ethereum address
- ▼ A name
- ▼ Additional Information in the form of a document (JSON)

▼ Each **Handler** is uniquely identified by the Ethereum address, thus addition of **Handler** requires no previous existence. **Handlers** add **Products** or handle **Products** and add **Checkpoints**. **Handlers** are registered by the **Owner** of the contract.

Design of the Handler

```
▼ struct handler {  
    bytes32 name;  
    bytes32 addInfo;  
    existence exists;  
}
```

- ▼ where `existence` is an enumeration required to check the existence of a structure in a mapping: `enum existence {NO,YES}`
- ▼ Members `name`, `addInfo` retain text encoded in `bytes32`. `Bytes32` cost less since solidity is optimized to process 32 bytes. Thus, any other type (`string`, `bytes11`, `bytes`, ...) require conversion which affects required gas.
- ▼ Moreover, inter-contract communication cannot be performed with `Strings`. However, `bytes32` encoded `Strings` can be communicated.

Properties of a Product

- ▼ A product is characterized by the following properties:
 - ▼ Unique ID (**uid**)
 - ▼ Additional Info in the form of a document (JSON)
 - ▼ **Checkpoints** indicating the locations that **Handlers** have interacted with the product
- ▼ The addition of a product can be performed only by a **registered Handler** and the product **uid** should be **unique**. Addition of a product also implies the addition of the initial **Checkpoint**.

Design of a Product

```
▼ struct product {  
    bytes32 uid;  
    bytes32 addInfo;  
    mapping(uint256 => checkpoint) checkpointArray;  
    uint256 numOfCheckpoints;  
    existence exists;  
}
```

- ▼ The `existence` type member is an enumeration required to check the existence of a structure in a mapping: `enum existence {NO,YES}`
- ▼ The member `checkpointArray` retains all checkpoints. Use of an array would result in an error during compilation since assignment of structs is not yet supported (when adding a product). Workarounds exist but substantially increase amount of operations and thus cost. The member `numOfCheckpoints` retains maximum number of records to enable fast access.

Definition of a Checkpoint

- ▼ A **Checkpoint** retains information of an interaction between a product and a handler. Thus, a **Checkpoint** includes the following:
 - ▼ The Ethereum address of the handler
 - ▼ The latitude and longitude (coordinates) where the interaction took place
 - ▼ Additional info in the form of a document (JSON)
 - ▼ Timestamp denoting when the interaction took place
- ▼ **Checkpoints** are retained in an array implemented with a mapping inside each **Product**. **Checkpoints** can be added only if the **Handler** and the **Product** both exist.

Definition of a Checkpoint

```
▼ struct checkpoint {  
    address handler;  
    uint256[2] coord;  
    bytes32 addInfo;  
    uint256 timestamp;  
}
```

▼ Solidity supports only integers thus coordinates have to be multiplied by 10^{10} to cast them to integers of adequate accuracy.

Common Elements of the three structures

- ▼ The three structures share some common elements such as generic fields of type bytes32. These generic fields can be used to share different types of characteristics allowing flexibility of use.
- ▼ The bytes32 fields can be extended into arrays for storing a larger set of properties in the form of document (JSON)
- ▼ The **Handler** and **Product** are stored in mappings from *Key* to *Struct*. Mappings allow for efficient random access of elements as well as an efficient mechanism to check uniqueness and existence based on key.
- ▼ However, elements in mappings cannot be accessed sequentially. Thus, in order to retrieve all structures in a mapping an *Iterable Mapping* should be used.

Iterable Mapping

- ▼ An *Iterable Mapping* is a data structure formed by a *mapping* between *keys* and *structs* and an *array* of keys.
- ▼ The *array* of *keys* is used in order to retain all stores *keys* in the *mapping*. *Arrays* are data structures that can be traversed sequentially.
- ▼ Traversing the *array* of *keys* **K** results in traversing all stored *structs* in mapping **M** as **M[K[i]]**.
- ▼ Mapping also allows for random access to the *structs* based on a key, thus an *Iterable Mapping* combines the advantages of *mappings* and *arrays*. For **Handler** and **Product** *structs* we have:

```
mapping(bytes32 => product) productIndex;  
bytes32[ ] productKeys;
```

```
mapping(address => handler) handlerIndex;  
address[ ] handlerKeys;
```

Addition of Handlers, Products and Checkpoints

- ▼ Addition of **Handlers** should enforce uniqueness. Checking uniqueness can be performed by examining the return value of a *mapping*. Initially all values of a *mapping* are considered to be **0 (NULL)**.
- ▼ However, the condition (**M [K] == 0**) is not applicable when *mapping* points to structures. In practice, the aforementioned condition is 'safely' applicable only in case of *mappings* to *uint*.
- ▼ Thus, the *existence* type variable is required to check for previous record on a *mapping* based on a *Key*. Checking *existence* then is of the form:

```
require(handlerIndex[handlerAddress].exists != existence.YES);
```
- ▼ A syntax:

```
require(handlerIndex[handlerAddress].exists == existence.NO);
```
- ▼ will **NOT work** since an initial value '**0**' of a *mapping* to a *struct* does not imply that members will be **0**. Moreover, '**0**' of type *struct* cannot be cast and in consequence checked.

Addition of Handlers, Products and Checkpoints

```
function addHandler(address handlerAddress, bytes32 handlerName, bytes32 handlerInfo) public  
onlyOwner {  
    require(handlerIndex[handlerAddress].exists != existence.YES);  
    handler memory currentHandler;  
    currentHandler.name = handlerName;  
    currentHandler.addInfo = handlerInfo;  
    currentHandler.exists = existence.YES;  
    handlerIndex[handlerAddress] = currentHandler;  
    handlerKeys.push(handlerAddress);  
}
```

structs are initialized in memory before assigned to storage

keys are pushed to the array (*Iterable Mapping*)

Addition of Handlers, Products and Checkpoints

- ▼ The modifier **onlyOwner** denotes that only the *owner* of the contract can add *Handlers*. This modifier is part of the Owned contract. This contract is inherited by the *SupplyChain contract*.
- ▼ An *event* is emitted every time the owner changes
- ▼ contract supplyChain is Owned { ... }

```
contract Owned {  
    address public owner;  
    event OwnershipTransferred(address indexed _from, address indexed _to);  
    function Owned() public {  
        owner = msg.sender;  
    }  
    modifier onlyOwner {  
        require(msg.sender == owner);  
        _;  
    }  
    function transferOwnership(address _newOwner) public onlyOwner {  
        emit OwnershipTransferred(owner, _newOwner);  
        owner = _newOwner;  
    }  
}
```


Addition of Handlers, Products and Checkpoints

- ▼ The addition of a **Product** requires no prior existence, similarly to the procedure of adding a **Handler**. Additionally, each product has to be added by a registered **Handler**.
- ▼

```
require(handlerIndex[msg.sender].exists == existence.YES);  
bytes32 productAddress = keccak256(abi.encode(uid));  
require(productIndex[productAddress].exists != existence.YES);
```
- ▼ The **Product** address is formed through hashing of the *uid* using *keccak256* method. Hashing is used to enforce a standard length and type to uniquely identify products in a mapping. The *abi.encode* is required to encode *bytes32* to *bytes*, which is accepted by the method.
- ▼ Addition of a **Product**, implies also addition of a **Checkpoint** along with related info, since it is considered as a point of creation for the **Product**.

Addition of Handlers, Products and Checkpoints

```
function addProduct(bytes32 uid,bytes32 productInfo,bytes32
checkpointInfo,uint256 lat,uint256 lon) public {
    require(handlerIndex[msg.sender].exists == existence.YES);
    bytes32 productAddress = keccak256(abi.encode(uid));
    require(productIndex[productAddress].exists != existence.YES);
    product memory currentProduct;
    checkpoint memory currentCheckpoint;
    currentProduct.uid = uid;
    currentProduct.addInfo = productInfo;
    currentProduct.numOfCheckpoints = 0;
    currentProduct.exists = existence.YES;
```

```
    currentCheckpoint.handler = msg.sender;
    currentCheckpoint.addInfo = checkpointInfo;
    currentCheckpoint.coord[0] = lat;
    currentCheckpoint.coord[1] = lon;
    currentCheckpoint.timestamp = now;
    productIndex[productAddress] = currentProduct;
    productKeys.push(productAddress);
    product storage tmpProduct = productIndex[productAddress];
    tmpProduct.checkpointArray[tmpProduct.numOfCheckpoints]
= currentCheckpoint;
    tmpProduct.numOfCheckpoints++;
}
```

*pointer to avoid
long names*

Addition of Handlers, Products and Checkpoints

- ▼ The addition of a **Checkpoint** requires existence of **Handler** and **Product**:
 - ▼ `require(handlerIndex[msg.sender].exists == existence.YES);`
 - ▼ `bytes32 tmpIndex = keccak256(abi.encode(uid));`
 - ▼ `require(productIndex[tmpIndex].exists == existence.YES);`
- ▼ **Checkpoints** are retained in an *Iterable Mapping* with a *uint* key, incremented with the addition of each **Checkpoint**. This approach is utilized to avoid compilation errors triggered by the *addProduct* function, since arrays cannot be assigned yet.
- ▼ **Note:** We always check existence against `existence.YES`, since `existence.NO` implies prior initialization in the deployment of the contract for a particular structure, which (apart from *uints*) is not performed.

Addition of Handlers, Products and Checkpoints

```
function addCheckpoint(bytes32 uid, bytes32 checkpointInfo, uint256 lat, uint256 lon) public {  
    require(handlerIndex[msg.sender].exists == existence.YES);  
    bytes32 tmpIndex = keccak256(abi.encode(uid));  
    require(productIndex[tmpIndex].exists == existence.YES);  
    checkpoint memory currentCheckpoint;  
    currentCheckpoint.handler = msg.sender;  
    currentCheckpoint.addInfo = checkpointInfo;  
    currentCheckpoint.coord[0] = lat;  
    currentCheckpoint.coord[1] = lon;  
    currentCheckpoint.timestamp = now;  
    tmpProduct.checkpointArray[tmpProduct.numOfCheckpoints] = currentCheckpoint;  
    tmpProduct.numOfCheckpoints++;  
}
```

Getters

- ▼ **Getters** are **constant (view or pure lately)** type functions in a Smart Contract that are used in order to return data.
- ▼ **Getters** are not considered as transactions, thus they do not require payment. However, there is a limit on how many local variables (including returning ones) can be used (similar to transactions). This limit is 16 else there is a compilation error (stack too deep).
- ▼ Thus a good practice is to keep **getters** focused and simple. **Getters** cannot be arbitrarily complex, since 'gas' is required also for them (local limit). In the case of **getters** 'gas' is not charged and act as a computational limit for the local node.
- ▼ Calling **getters** multiple times is a fast procedure since they traverse a local copy of the chain.

Getters

▼ The **getters** in the application are:

- ▼ **getProduct**: Returns all information and **checkpoints** related to a **product**.
- ▼ **getHandler**: Returns all information related to a **handler**.
- ▼ **getAllProducts**: Returns all information related to all product. This function *does not* return the **checkpoints**.
- ▼ **getAllHandlers**: Returns all information related to all registered **handlers**.

▼ Structs are returned as tuples in Solidity ([{...}, {...}, ...]).

Get Product

```
function getProduct(bytes32 uid) public constant
returns(bytes32,address[],bytes32[],uint256[],uint256[],uint256[]) {
    bytes32 tmpIndex = keccak256(abi.encode(uid));
    require(productIndex[tmpIndex].exists == existence.YES);
    product storage tmpProduct = productIndex[tmpIndex];

    address[] memory handlers = new address[](tmpProduct.numOfCheckpoints);
    bytes32[] memory checkpointInfos = new bytes32[](tmpProduct.numOfCheckpoints);
    uint256[] memory lats = new uint256[](tmpProduct.numOfCheckpoints);
    uint256[] memory lons = new uint256[](tmpProduct.numOfCheckpoints);
    uint256[] memory timestamps = new uint256[](tmpProduct.numOfCheckpoints);

    for(uint i = 0; i < tmpProduct.numOfCheckpoints; i++) {
        handlers[i] = tmpProduct.checkpointArray[i].handler;
        checkpointInfos[i] = tmpProduct.checkpointArray[i].addInfo;
        lats[i] = tmpProduct.checkpointArray[i].coord[0];
        lons[i] = tmpProduct.checkpointArray[i].coord[1];
        timestamps[i] = tmpProduct.checkpointArray[i].timestamp;
    }
    return (tmpProduct.addInfo,handlers,checkpointInfos,lats,lons,timestamps);
}
```

Existence of
product

Storage pointer to
avoid long – names
and dereferencing

Memory arrays
variables to hold
checkpoints

Populate arrays

Get Handler

- ▼ **function** `getHandler(address addressOfHandler)` **public constant** **returns(bytes32,bytes32)** {
 `require(handlerIndex[addressOfHandler].exists == existence.YES);`

 `return(handlerIndex[addressOfHandler].name,handlerIndex[addressOfHandler].addInfo);`
}
- ▼ Returning info related to a **Handler** requires its prior existence.
- ▼ The rest of the **getters** are similar.
- ▼ As a rule of **thumb**: Returning arrays of elements stored in mappings requires definition of **memory** type arrays as intermediate storage. Elements stored in arrays can be returned as is.
- ▼ Moreover, variable length arrays and variables can be returned simultaneously, since **getters** can return tuple of tuples.

The *supplyChain* smart contract

```
pragma solidity ^0.4.24;
contract Owned { ... }
contract supplyChain is Owned {
    enum existence {NO,YES}
    struct handler { ... }
    struct checkpoint { ... }
    struct product { ... }

    mapping(address => handler) handlerIndex;
    address[] handlerKeys;

    mapping(bytes32 => product) productIndex;
    bytes32[] productKeys;

    function addHandler(address handlerAddress,bytes32 handlerName,bytes32 handlerInfo) public onlyOwner { ... }
    function addProduct(bytes32 uid,bytes32 productInfo,bytes32 checkpointInfo,uint256 lat,uint256 lon) public { ... }
    function addCheckpoint(bytes32 uid,bytes32 checkpointInfo,uint256 lat,uint256 lon) public { ... }

    function getProduct(bytes32 uid) public constant { ... }
    function getHandler(address addressOfHandler) public constant returns(bytes32,bytes32) { ... }
    function getAllProducts() public constant returns(bytes32[],bytes32[],uint256[]) { ... }
    function getAllHandlers() public constant returns(address[],bytes32[],bytes32[]) { ... }
}
```

Single vs Multi Contract approach

- ▼ In general there is **no** better approach when designing smart contracts.
- ▼ A **key** factor when selecting an approach is **cost** per transaction, especially when Ethereum is considered.
- ▼ In a multi-contact approach each **Product** is represented by a contract, with its address being the *uid*.
- ▼ Multi-contract approach reduces the amount of code required and enables easy deletion from the chain through **`_selfdestruct()`**.
- ▼ Deploying a **Smart Contract** is more expensive than performing a **Transaction**.
- ▼ Single contract approach enables easier implementation of user access control. Moreover, collecting and returning data is easier.
- ▼ Which of the two approaches is generally better in our case? **NONE, Each type has its advantages and disadvantages.**

Deployment of contract

- ▼ The contract can be deployed using a variety of tools:
 - ▼ *Remix*: Online compiler and deployer. Requires Metamask in order to sign transaction for contract deployment.
 - ▼ *Truffle*: A collection of tools, allowing for deployment of multiple contracts with complex dependencies. For local nodes requires unlocking. For remote nodes, it requires *truffle-hdwallet-provider*.
 - ▼ Custom scripts in *python* or *node.js*. Requires programming of methods and functions to read the contract, compile it, estimate gas, deploy and wait for receipt. Custom scripts are a good practice for very complex automated deployments.
- ▼ We have a single contract, thus the deployment is relatively straight forward using Remix (<https://remix.ethereum.org/>). We will deploy in the Ropsten test network.

Deployment of contract - Remix

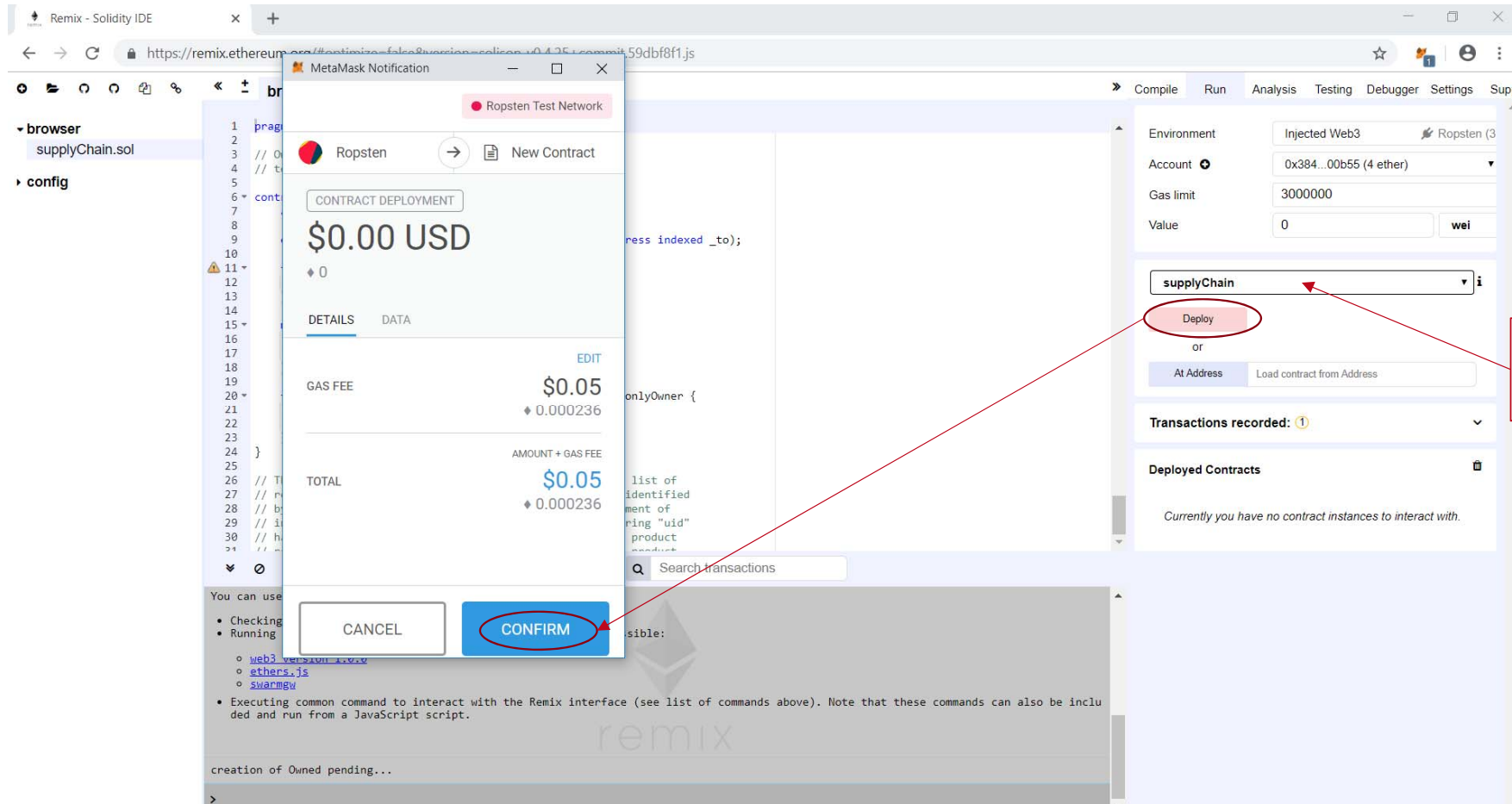
The screenshot displays the Remix IDE interface. The main editor shows a Solidity contract named `supplyChain.sol` with the following code:

```
1 pragma solidity ^0.4.24;
2
3 // Owned contract is used to restrict some actions
4 // to the owner of the contract
5
6 contract Owned {
7     address public owner;
8
9     event OwnershipTransferred(address indexed _from, address indexed _to);
10
11     function Owned() public {
12         owner = msg.sender;
13     }
14
15     modifier onlyOwner {
16         require(msg.sender == owner);
17         _;
18     }
19
20     function transferOwnership(address _newOwner) public onlyOwner {
21         emit OwnershipTransferred(owner, _newOwner);
22         owner = _newOwner;
23     }
24 }
25
26 // The contract is used to track product journey through a list of
27 // registered handlers in the supply chain. Handlers are identified
28 // by their Ethereum address, name and a chosen JSON document of
29 // information. Each product is identified by a unique string "uid"
30 // hashed in order to be used as a key to a mapping. Each product
31 // contains a list of actions performed by handlers on the product
```

The right-hand panel shows the compilation options. The **Run** tab is selected, and the **Start to compile** button is highlighted with a red circle. The current compiler version is `version:0.4.25+commit.59dbf8f1.Emscripten.clang`. Below the compiler options, the contract `Owned` is selected, and the **Details** button is visible. A warning message is displayed: "Static Analysis raised 15 warning(s) that requires your attention. Click here to show the warning(s)." The bottom panel shows the terminal with the following text:

```
- Welcome to Remix v0.7.3 -
You can use this terminal for:
• Checking transactions details and start debugging.
• Running JavaScript scripts. The following libraries are accessible:
  o web3 version 1.0.0
  o ethers.js
  o swarmgw
• Executing common command to interact with the Remix interface (see list of commands above). Note that these commands can also be included and run from a JavaScript script.
```

Deployment of contract - Remix



Deployment of contract

- Contract is deployed at [0xfb4a4fcc38f91596d7ec25f8834dd79077bcef59](#)
- We can interact with the contract through Remix.
- The ABI, required for interacting with the contract, and BIN, required to manually deploy the contract, can be extracted from Remix. However, copying and pasting manually in JSON files might result in problems related to encoding.
- Thus, producing the ABI using **solc** compiler is suggested. The solidity compiler (solcjs) can be installed using **npm** (node.js package manager) or a Linux package manager. The installation (with npm) command is: **npm install -g solc**
- The ABI can be extracted as:
solcjs --abi --bin supplyChain.sol
- This command outputs 4 files since the Solidity file includes 2 contracts. The ***.abi** files are JSON files including the ABI for each contract (Owned,supplyChain).
- In order to interact with the *supplyChain* contract the corresponding ABI file is required.

Deployment of contract - Python

- ▼ Python can be used to design custom deployment scripts for Solidity contracts.
- ▼ However, the procedure is not straight forward and required development of functions and tools to compile and deploy the contract.
- ▼ Requirements:
 - ▼ python3.6 and libraries web3, py-solc
 - ▼ pip3.6 install web3
 - ▼ pip3.6 install py-solc
 - ▼ solc compiler above v0.4.24
 - ▼ python3.6 -m solc.install v0.4.24
 - ▼ An full node url (RPC interface). This can be easily obtained by registering at Infura (<https://infura.io>) and obtaining a token for Ethereum mainnet or any testnet such as Ropsten. The url should look like `https://ropsten.infura.io/<token>`

Deployment of contract - Python

▼ The script begins as follows:

```
▼ import web3
import time
import sys
import os
from solc import compile_source
from web3 import Web3, HTTPProvider
```

```
node_url = 'https://ropsten.infura.io/<token>'
privateKey = '...'
publicKey = '...'
full_contract_path = './supplyChain.sol'
gasPrice = 2
chainId = 3
```

Token is required in order to establish connection to the Blockchain

Private Key and Public Address starting with 0x...

Path to the contract, gas price in Gwei and chain ID of the Ethereum network

Deployment of contract - Python

- ▼ Connection to a node is established using the following function, which returns a proper object:
- ▼

```
def con2ETH(full_node_url):  
    t_w = Web3(HTTPProvider(full_node_url))  
    return t_w
```
- ▼ Reading and compiling the contract is performed using another function:
- ▼

```
def loadAndCompile(contract_path):  
    with open(contract_path) as f:  
        file = f.read()  
        compiled_sol = compile_source(file);  
    return (compiled_sol)
```

Deployment of contract - Python

These functions are used in order to deploy the contract using the following function:

def processAndDeployContract(full_node_url,contract_path,public_add,private_key,gasPrice,chainID):

```
w3 = con2ETH(full_node_url)
cntrct = loadAndCompile(full_contract_path)
cntrct_interface = cntrct['<stdin>:supplyChain']
cntrct_abi = cntrct_interface['abi']
cntrct_bin = cntrct_interface['bin']
cntrct_instance_full = w3.eth.contract(abi=cntrct_abi, bytecode=cntrct_bin)
nonce = w3.eth.getTransactionCount(public_add);
gas = cntrct_instance_full.constructor().estimateGas({'from':public_add})
txn = cntrct_instance_full.constructor().buildTransaction({'chainId':chainID,'gas':gas,'gasPrice':
w3.toWei(gasPrice,'gwei'),'nonce':nonce})
signed_txn = w3.eth.account.signTransaction(txn,private_key = private_key)
w3.eth.sendRawTransaction(signed_txn.rawTransaction)
txn_hash = w3.toHex(w3.sha3(signed_txn.rawTransaction))
receipt = wait_for_receipt(full_node_url,txn_hash,2)
deployed_contract_address = receipt['contractAddress']
return deployed_contract_address
```

Connect to Ethereum, load and compile the contract

Extract ABI and BIN for the main contract and form a contract instance

Estimate Gas by simulating the transaction and build the transaction

Sign, Deploy and Wait for the receipt to derive the contract address

Deployment of contract - Python

- Waiting on a receipt requires polling the Ethereum Blockchain regularly:
- ```
def wait_for_receipt(full_node_url,txn_hash,poll_interval):
 w3 = con2ETH(full_node_url)
 while True:
 txn_receipt = w3.eth.getTransactionReceipt(txn_hash)
 if txn_receipt:
 return txn_receipt
 time.sleep(poll_interval)
```
- The script can be deployed by:  

```
python3.6 deployer.py
```
- The result is the address of the smart contract. Python can be used to setup various tools for interacting with the Blockchain as well as more advance procedures such as servers, etc.

# Inheritance in Solidity Smart Contracts

---

- ▼ In Solidity inheritance works similarly to any other object oriented language. The keyword “is” is used:

```
contract name1 { ... }
contract name2 { ... }
```

```
contract derived is name1,name2 { ... }
```

- ▼ The inherited contracts are listed from the “most base-like” to the “most derived”. This order is used to resolve conflicts arising with functions of the same name within contracts.
- ▼ All function calls are virtual, leading to invocation of the “most derived” function, for inherited function sharing the same name. This feature can be bypassed by invoking explicitly the contract name i.e: <Contract\_name>.<function\_name>(input\_1,...,input\_n)
- ▼ In practice, on the blockchain a single contract is created. The contract that inherits multiple contracts retains a copy of the code of all the inherited contracts.

# Inheritance in Solidity Smart Contracts

---

- ▼ Within each contract, a function can be characterized as:
  - ▼ external (interface functions): External functions can be called through transactions or from other contracts. This functions cannot be called internally (however `this.function_name()` can be used, which is an expensive CALL type operation).
  - ▼ public (interface functions): This functions and variables can be called either internally or via transactions. These functions are more expensive than external because they copy array arguments to memory. Storing to memory is an expensive operation. Variables denoted public are automatically assigned a getter.
  - ▼ internal: Denotes functions or variables that can be accessed from within the contract or contracts deriving from it.
  - ▼ private: Function and variables defined as private are only accessible within the contract they are defined and not the inherited contracts.
- ▼ Inherited contracts can reside in different files and invoked through the “import” command, i.e.:  
  
`import “./test.sol”`
- ▼ A large collection of Smart Contracts is included in the OpenZeppelin collection.

# Inheritance in Solidity Smart Contracts

---

- ▼ The OpenZeppelin (<https://github.com/OpenZeppelin/openzeppelin-solidity/tree/master/contracts>) Smart Contract collection can be used to design complex contracts through inheritance. This collection has Smart Contracts for:
  - ▼ Access control
  - ▼ Token Crowdsale
  - ▼ Cryptography
  - ▼ Math operations
  - ▼ Ownership
  - ▼ Payments
  - ▼ Lifecycle operations
  - ▼ Token related utilities and contracts
  - ▼ Examples
- ▼ GitHub has a very large collection of Smart Contracts developed by a lot of programmers in the Ethereum community, including contracts developed by the founder of the Ethereum Crypto Currency.

# Libraries in Solidity Smart Contracts

---

- Solidity includes the notion of a “library”. A “library” is a special type of contract that does not allow “payable” and fallback functions:

```
library name1 { ... }

contract name2 {
 function f1 {
 name1.f();
 }
}
```

- They are considered as implicit base contracts of the contracts that use them (linked in the bytecode level, with a placeholder logic). Thus, they can be invoked directly and need **NOT** be included in the inherited contracts list following the “is” keyword.
- Calling functions of libraries results in invoking a specialized instruction (DELEGATECALL) passing the call to the library seamlessly executing as if it was a function of the contract itself.

# Libraries in Solidity Smart Contracts

---

- ▼ Libraries do not have their own storage, but they can modify the storage of the calling contract.
- ▼ Function inside libraries can be easily included into contracts with the keyword “using”:

```
contract name1 {
 using library_name for data_type;
}
```

- ▼ The keyword “using” gives all functions of the library, that take first argument of type *data\_type*, to that *data\_type*. The wildcard “\*” can be used for “data\_type” to inherit all methods in the Library to the corresponding types.
- ▼ Libraries can include structs or events and can be imported from other Solidity files:  
`import { library_name } from “./filename.sol”;`
- ▼ Libraries can be used for minimizing repetitive chunks of code, or extend functionality of certain data types such as *uint*.



# Libraries in Solidity Smart Contracts

---

- ▼ Libraries can be used to modify the storage of their linked contract. In order to do that the this variables should be given as input and characterized with the keyword “storage”.
- ▼ The OpenZeppelin makes heavy use of libraries in the available Solidity contracts collection.
- ▼ One of the most popular and widely used library is the SafeMathLib, which extends *uint* operations fortifying against overflows, underflows or errors related to conversions.
- ▼ Another major issue of libraries (until recently) was the unavailability of an upgrade mechanism. Upgrading a library resulted in redeployment of a contract. This is an unwanted feature since it affects cost.
- ▼ This major issue was tackled by OpenZeppelin with the introduction of proxy libraries, that allow for updating, without redeployment of contracts.
- ▼ Proxy libraries allow for performing essential (usually security related) changes without modifying the contract.
- ▼ **An import suggestion for developers is to try to incorporate as much code as possible from widely used deployed libraries since they usually are audited and vetted by the community.**

# Events in Solidity Smart Contracts

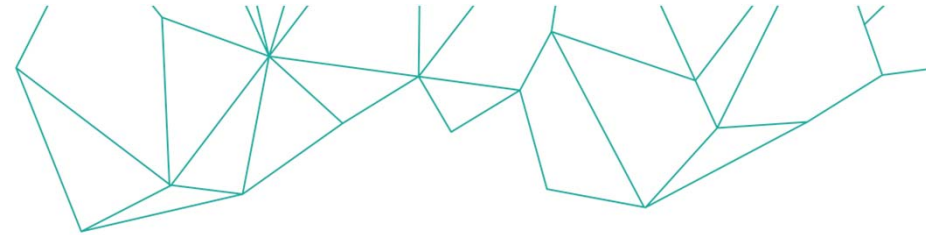
---

- ▼ An “event” in Solidity is a convenient way for accessing the Ethereum Virtual Machine logging facilities.
- ▼ Events behave as members of a contract and thus can be inherited.
- ▼ Events are defined as follows:  
`event event_name(var1,var2,...,varn);`
- ▼ Events can log information as values of variables. This is why they are often used as a relatively “cheap” means of storage on the Ethereum Blockchain. Storing info by emitting an event is cheaper than storing it directly in the storage of a contract. However, information are not accessible from inside a contract.
- ▼ Emission of events is performed by invoking the command “emit” (in older versions of solidity, “emit” was not required):  
`emit event_name(var1,...,varn);`
- ▼ Events are logging information stored in the Ethereum blocks. They can be used to track certain actions performed by function of a smart contract.

# Events in Solidity Smart Contracts

---

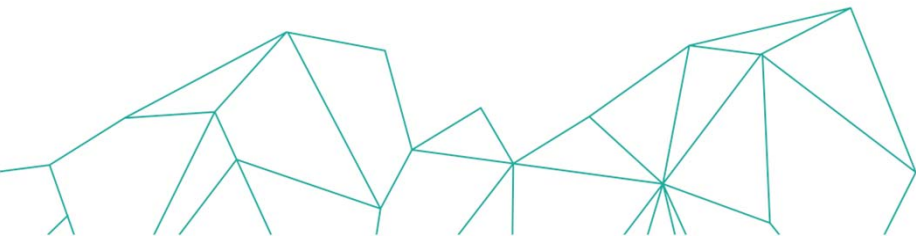
- ▼ Watching for events can be performed by the `.watch` method offered by `web3.js` (<v1.0) or `web3.py`.
- ▼ In later versions, subscription to a full node and definition of a filter is required. A filter is used to “filter” events based on “indexed” parameters.
- ▼ The “indexed” parameters are given upon definition of an event:  
`event event_name(uint256 indexed val)`  
The parameter “val” can be used as a filter.
- ▼ Listening for a large number of events is a computationally intensive procedure. Subscribing to full nodes for listening to events very intensive procedures. This is why remote nodes (Infura) do not allow for subscription to events in traditional ways (RPC) and instead websocket type API is used for subscription.
- ▼ It should be noted that multiple events can be emitted by a contract.



## Decentralized Training Series Workshop: Certified Ethereum Specialist

19-20 November 2018, Athens, Greece

# II / HTML + JavaScript (Front End)



# Interacting with the Ethereum Blockchain and the Contract

---

- ▼ The distributed App (dApp) should interact with the Ethereum Blockchain. Interaction with the Ethereum Blockchain is usually performed using the *Web3* library.
- ▼ The *Web3* library is available in various languages including JavaScript (*web3.js*) and *Python* (*web3.py*).
- ▼ Interaction requires the following components (given in index.html):
  - ▼ Web3: For interacting with the deployed smart contract  
<https://cdn.jsdelivr.net/gh/ethereum/web3.js/dist/web3.js>
  - ▼ jQuery: For reading the ABI of the smart contract  
<https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js>
  - ▼ Google Maps Javascript API: For visualizing geographical data of checkpoints  
[https://maps.googleapis.com/maps/api/js?key=<API\\_TOKEN>](https://maps.googleapis.com/maps/api/js?key=<API_TOKEN>)
  - ▼ index.js: Custom set of tools described in the following to interact with the contract

## Connecting with Ethereum Blockchain

---

- ▼ Connection to the Ethereum Blockchain is performed as follows:  
*var web3 = new Web3(web3.currentProvider);*
- ▼ The struct *web3.currentProvider* is injected by MetaMask. In practice the injected *web3* structure offers a Gateway to Ethereum through Infura.
- ▼ Moreover, MetaMask acts also as a control mechanism allowing for rejecting transactions, when triggered. MetaMask retains a log of all transactions.
- ▼ Other methods for connecting to the Ethereum Blockchain are:
  - ▼ *var web3 = new Web3("<url\_to\_an\_ethereum\_node>");*
  - ▼ Websocket connection as well as ipc type connection are also supported.

## Forming a contract instance with *web3*

---

- ▼ A contract instance is formed by two components: (i) the contract's ABI, (ii) the contract's address.
- ▼ Reading the contract's ABI with jQuery (sync):

```
var response = jQuery.get({url:'contracts/abi/supplyChain_sol_supplyChain.abi',
 async:false,
 success:function(data) { }
 });
var abi = JSON.parse(response.responseText);
```
- ▼ The ABI is parsed as a JSON document and is used with *web3*:

```
var supplyChain = web3.eth.contract(abi);
var contractInstance = supplyChain.at('0xda3f941dc9e9e69ea3793b8f475dfbe843cfacb9');
```
- ▼ Without the deployment address, *web3* cannot interact with the Ethereum Blockchain.
- ▼ *contractInstance* includes all methods of the contract, described in Solidity.

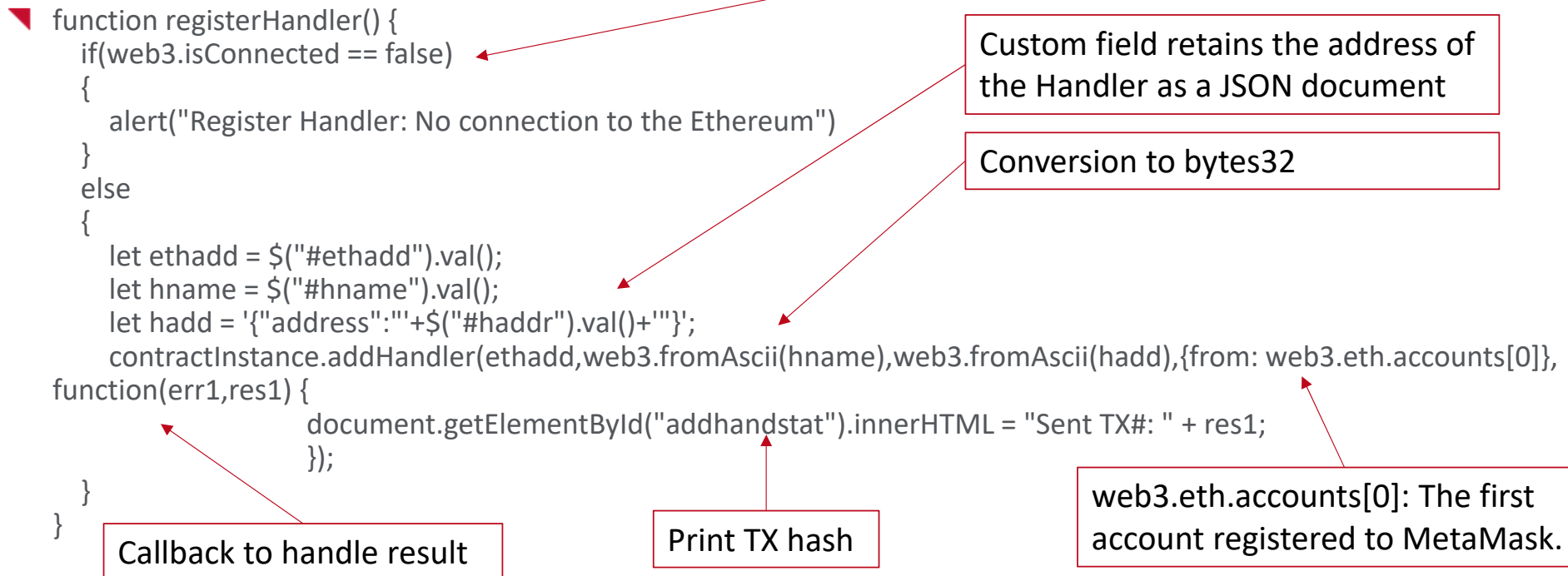
## Transactions related to the contract

---

- ▼ In practice the contract instance retains the signatures for all methods. A signature is defined as the name of the function in the contract along with the data types of inputs. For example:  
**`addProduct(bytes32,bytes32,bytes32,uint256,uint256)`**
- ▼ The signature is hashed and the first 4 bytes (of the ASCII form) are used as method ID, in order to uniquely define the method, followed by the encoded parameters (bytes). This payload is stored in the *data* part of the transaction. More information on the form of a transaction can be found in the Ethereum documentation: <https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI#examples>.
- ▼ This procedure is performed automatically by the *web3* library.



# Registration of a Handler using JavaScript



## Registration of a Handler using JavaScript

---

- ▼ The custom field allows for extending functionality since it can handle anything as long as its in JSON document format. Thus, use of custom fields that can store JSON formatted documents improves flexibility.
- ▼ The cost of the transaction (gas) as well as the (gas price) are estimated automatically by MetaMask.
- ▼ The total gas amount is estimated also by *web3*. The estimation is performed by simulation of the transaction on the Ethereum Virtual Machine (EVM). The Gas Limit can be computed as:
$$G = G_{transaction} + G_{datanonzero} * dataLengthBytes = 21000 + 68 * dataLengthBytes$$
- ▼ Simulation is performed by the *web3.eth.estimateGas(txn)*. In case of a failing transaction the *estimateGas* function returns an error, thus avoiding costs related to submission of a failing transaction.
- ▼ This procedure is automatically performed by MetaMask, warning the user for a failing transaction.

## Registration of a Handler using JavaScript

---

- ▼ An example of gas estimation for the *addHandler* contract method is as follows:

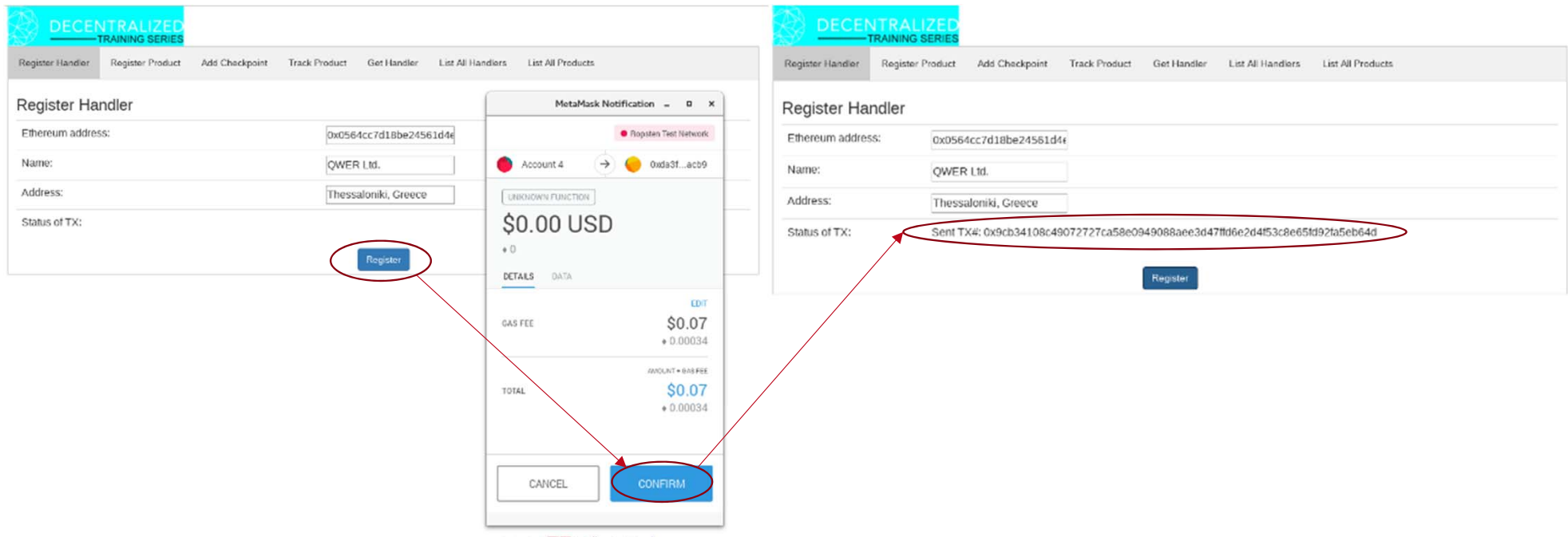
```
contractInstance.addHandler.estimateGas(ethadd,web3.fromAscii(hname),web3.fromAscii(hadd),{from:
web3.eth.accounts[0]}, function(err1,res1) {
 console.log(res1);
});
```
- ▼ The transaction can be performed, inside the *callback*, with the estimated amount of Gas (res1).
- ▼ The estimation of Gas is not mandatory in the case of MetaMask, since it is performed by it. However, in the case of custom wallets or handling of transactions without MetaMask, *estimateGas* is required in order to avoid failing transactions, which increase cost, or prescribing insufficient amounts of gas.
- ▼ The cost of a transaction is the product of *gas* and the *gas price*. Estimation of the Gas price is not straightforward. *Web3* has methods that analyze the *gas price* of the transactions of previous blocks or obtain *gas price* from a constituent node. *Gas price* can be obtained from observatories such as EthGasStation.
- ▼ Estimation of the *gas price* is a computationally intensive procedure.

## Registration of a Handler using JavaScript

---

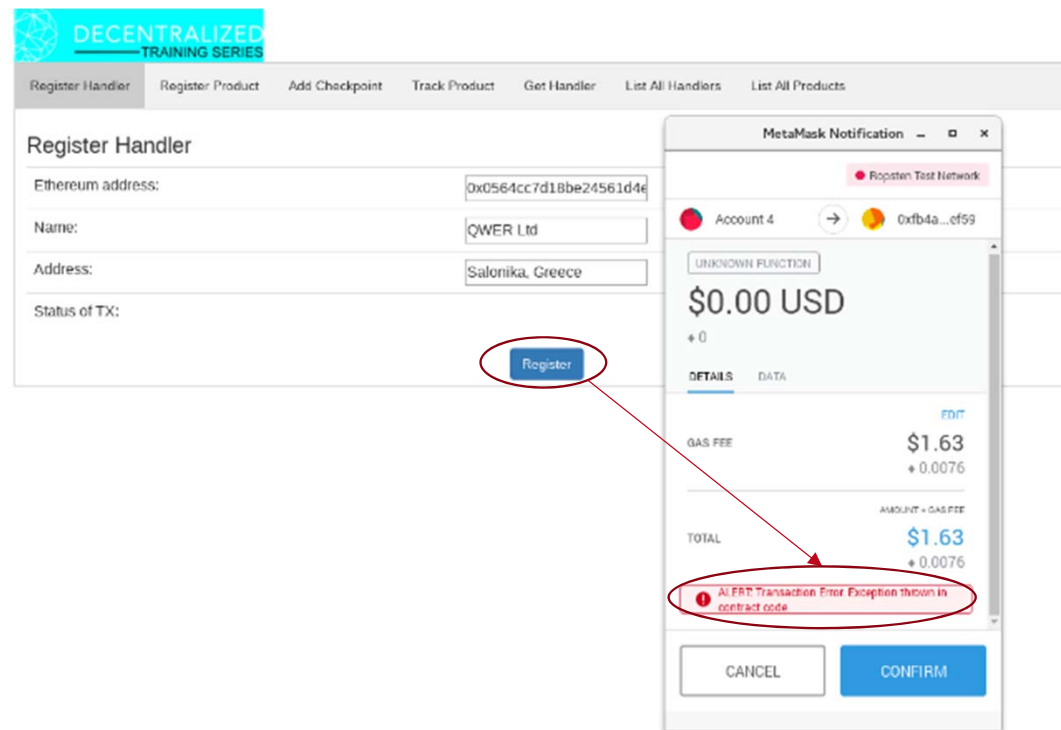
- ▼ Traversing and analyzing cost of transactions requires a substantial amount of time especially for large number of blocks.
- ▼ On the other hand estimation by the price given by a *constituent* full node might be of low accuracy since this estimate is formed by the transactions passing through that specific node which probably is a small fraction of the overall number of transaction.
- ▼ The safest choice for selecting the *gas price* is obtaining it from a service that is dedicated to analyzing *gas price* such as EthGasStation.
- ▼ The *gas price* affects the amount of time required until the transaction has been confirmed on the network, since miners usually serve transaction with higher *cost* first.

# Registration of a Handler using JavaScript



## Repeating registration of a Handler using JavaScript

- Addition fails due to the fact that address is registered to other handler. Thus, duplication of address is not allowed by the Smart Contract.



# Registration of a Product by Handler in JavaScript

- ▼ The registration of a Product by a handler is performed by invoking the following function:

```
function registerProduct() {
 if(web3.isConnected == false) {
 alert("Register Product: No connection to the Ethereum")
 } else {
 let uid = $("#uid").val();
 let prodInfo = '{"name":"' + $("#name").val() + '"}';
 let chkInfo = '{"action":"' + $("#action").val() + '"}';
 let lat = parseFloat($("#lat").val()) * Math.pow(10,10);
 let lon = parseFloat($("#lon").val()) * Math.pow(10,10);

 contractInstance.addProduct(web3.fromAscii(uid),web3.fromAscii(prodInfo),web3.fromAscii(chkInfo),lat,lon,{from
: web3.eth.accounts[0]}, function(err1,res1) {
 document.getElementById("addprodstat").innerHTML = "Sent TX#: " + res1;
 });
 }
}
```

Check connection to Ethereum

Custom fields in product and checkpoint are used to store name of product and action performed at a certain checkpoint in JSON format

Conversion of Floating point numbers to integers by multiplication

Encoding to bytes32

Update the TX#

## Registration of a Product by Handler in JavaScript

---

- ▼ The contract is responsible for checking if the Handler invoking the method to register a Product is registered. Failure to satisfy uniqueness results in error and does not affect the contract.
- ▼ The latitude and longitude have to be parsed initially as floating point numbers before multiplication by  $10^{10}$ .
- ▼ The Unique ID can be any number up to 32 characters, since that is the maximum number fitting in a bytes32 type variable.
- ▼ The name and action of the first checkpoint are stored in JSON format, which is encoded in bytes32 according to the contract.



# Registration of a Product by Handler in JavaScript

The image shows a web application interface for the 'Decentralized Training Series' with a 'Register Product' form. The form includes fields for Unique ID (12), Name (Onions), Action (creation), Latitude (40.736851), and Longitude (22.920227). A 'Register' button is at the bottom. A MetaMask notification window is open in the center, showing a transaction for \$0.00 USD with a gas fee of \$0.08. The 'CONFIRM' button is highlighted. A red arrow points from the 'Register' button on the form to the 'CONFIRM' button in the MetaMask window. Another red arrow points from the 'CONFIRM' button to the 'Status of TX' field on the form, which now displays the transaction hash: 'Sent TX#: 0x9bfa7aad177aa8cae1e3318fc277aabea63c657f7e7a1262143630f88292559a'.

**Register Product**

Unique ID: 12

Name: Onions

Action: creation

Latitude: 40.736851

Longitude: 22.920227

Status of TX: Sent TX#: 0x9bfa7aad177aa8cae1e3318fc277aabea63c657f7e7a1262143630f88292559a

**MetaMask Notification**

Account 2 → 0xfb4a...ef59

UNKNOWN FUNCTION

\$0.00 USD

DETAILS DATA

GAS FEE \$0.08

TOTAL \$0.08

CANCEL CONFIRM

## Addition of a Checkpoint to a Product by a registered Handler

---

- ▼ The contract is responsible for checking the existence of a Handler or a Product, where the Checkpoint will be registered. In case of any of these two properties not being satisfied the transaction fails due to the 'infinite gas' requirements.
- ▼ Each Checkpoint includes geographical information in the form of coordinates. These coordinates need conversion to integers as performed previously.
- ▼ The Handler address is extracted by the *msg* variable of each transaction, which includes information such as address that performed the transaction as well as the amount of Ethereum sent in a regular transaction.
- ▼ The description of the checkpoint action is stored in a JSON document and encoded to bytes32.

# Addition of a Checkpoint to a Product by a registered Handler

```
function addCheckpoint() {
 if(web3.isConnected == false) {
 alert("Add Checkpoint: No connection to the Ethereum")
 } else {
 let uid = $("#cuid").val();
 let action = '{"action":"' + $("#caction").val() + '"}';
 let lat = parseFloat($("#clat").val()) * Math.pow(10,10);
 let lon = parseFloat($("#clon").val()) * Math.pow(10,10);
 contractInstance.addCheckpoint(web3.fromAscii(uid), web3.fromAscii(action), lat, lon, {from:
web3.eth.accounts[0]}, function(err1, res1) {
 document.getElementById("addchkpnt").innerHTML = "Sent TX#: " + res1;
 return (res1);
 });
 }
}
```

Check connection to Ethereum

Custom action performed at a certain checkpoint in JSON format

Conversion of Floating point numbers to integers by multiplication

Encoding to bytes32

Update the TX#

# Addition of a Checkpoint to a Product by a registered Handler

The image shows a web application interface for the 'DECENTRALIZED TRAINING SERIES'. The main form is titled 'Add Product Checkpoint' and contains the following fields:

- Unique ID: 12
- Action: reload
- Latitude: 40.30069
- Longitude: 21.78896
- Status of TX: Sent TX#: 0xc5247d4ce6b1776f3ba8e854c89c5f73ea6f9f689010e58963dfa52a47e9197d

A blue 'Add Checkpoint' button is located at the bottom right of the form. A MetaMask notification dialog is overlaid on the form, showing the transaction details for 'Account 3' on the 'Ropsten Test Network'. The dialog includes a 'CONFIRM' button and a 'REJECT' button. A red arrow points from the 'Add Checkpoint' button to the 'CONFIRM' button in the MetaMask dialog. Another red arrow points from the 'Status of TX' field to the transaction hash.

MetaMask Notification

Account 3 → 0xfb4a...ef59

CONFIRM

0 \$0.00

DETAILS DATA

GAS FEE 0.000199 \$0.04

AMOUNT + GAS FEE

TOTAL 0.000199 \$0.04

REJECT CONFIRM

## Tracking of Product based on its Unique ID

---

- ▼ Tracking of a Product includes fetching all Checkpoints from the Ethereum Blockchain and visualizing the result on a Map based on the registered coordinates.
- ▼ Retrieving data from the Ethereum Blockchain is not considered as a regular transaction since it does not require any Ether to be performed.
- ▼ Data is retrieved by traversing the local (to the full node) copy of the Blockchain. However, to avoid excessive computational work from a node, the Calls for retrieving data are limited also by a tunable *gas limit*.
- ▼ Solidity cannot return structures, instead *struct* members are returned as tuples.
- ▼ The coordinates are visualized using Google Maps Javascript API.

# Tracking of Product based on its Unique ID

---

- Each Checkpoint has a custom field used for retaining the action performed at each Checkpoint. However, solidity pads *bytes32* from the right with 0.
- Decoding *bytes32* encoded data obtained from the Blockchain requires removal of padding. Parsing the decoded data as a JSON document 'as is' results in error. Padding can be easily removed as follows:
- ```
function cleanPadding(bstr) {  
    let len = bstr.length;  
    let i = len;  
    while (i >= 1) {  
        if(bstr[i-1] != '0') {  
            break;  
        }  
        i--;  
    }  
    return(bstr.substring(0,i));  
}
```

Tracking of Product based on its Unique ID

- ▼ The data returned from the *getProduct* contract getter are as follows: (bytes32,array of addresses, array of bytes32,array of uint256,array of uint256,array of uint256).
- ▼ The length of the arrays coincides with the number of Checkpoints of a product. The first field of the tuple is a bytes32 encoded JSON document with the name of the product.
- ▼ The arrays retain the addresses of the Handlers, the actions performed on the Checkpoints, the Latitudes and Longitudes as well as the timestamps.
- ▼ Tuple of arrays has the form **result[field][entry]**.

Tracking of Product based on its Unique ID

```
contractInstance.getProduct(web3.fromAscii(uid), function(err1,res1) {  
  let numOfCheckpoints = res1[1].length;  
  let tmpJSON = JSON.parse(web3.toAscii(cleanPadding(res1[0])));  
  ...  
  let lats = new Array();  
  let lons = new Array();  
  let actions = new Array();  
  let clat = 0.0;  
  let clon = 0.0;  
  for(let i = 0; i < numOfCheckpoints; i++) {  
    tmpJSON = JSON.parse(web3.toAscii(cleanPadding(res1[2][i])));  
    let date = new Date(Number(res1[5][i])*1000);  
    lats.push(parseFloat(res1[3][i])/Math.pow(10,10));  
    lons.push(parseFloat(res1[4][i])/Math.pow(10,10));  
    actions.push(tmpJSON["action"]);  
    clat += lats[i];  
    clon += lons[i];  
    ...  
  }  
  clat /= numOfCheckpoints;  
  clon /= numOfCheckpoints;  
  ...  
});
```

The number of Checkpoints coincides with the length of one of the arrays.

Clean Padding, encode to ASCII and parse as JSON

Convert UNIX timestamp to Date

Convert to Floating Point numbers

Center of Mass is computed to center the Map correctly

... denote parts where html fields are updated and the Map is instantiated.

Tracking of Product based on its Unique ID

```
var map = new google.maps.Map(document.getElementById('map'), {
  zoom: 5,
  center: new google.maps.LatLng(clat,clon),
  mapTypeId: google.maps.MapTypeId.ROADMAP
});
var infowindow = new google.maps.InfoWindow();
var mark, i;
for (let i = 0; i < numOfCheckpoints; i++) {
  marker = new google.maps.Marker({
    position: new google.maps.LatLng(lats[i],lons[i]),
    map: map
  });
  google.maps.event.addListener(marker, 'click', (function(marker, i) {
    return function() {
      infowindow.setContent(String(i+1)+'.' +actions[i]);
      infowindow.open(map, marker);
    }
  })(marker, i));
}
```

Creation of a Map and instantiation to *map* element in the html document

Chosen *zoom*, *center* and *type* for the Map

Place markers on the map based on Lat / Lon for all Checkpoints

Place the *action* on each info window popping upon click of a marker

Tracking of Product based on its Unique ID

DECENTRALIZED
TRAINING SERIES

Register HandlerRegister ProductAdd CheckpointTrack ProductGet

Track Product

Unique ID:

Track Product

DECENTRALIZED
TRAINING SERIES

Register HandlerRegister ProductAdd CheckpointTrack ProductGet HandlerList All HandlersList All Products

Track Product

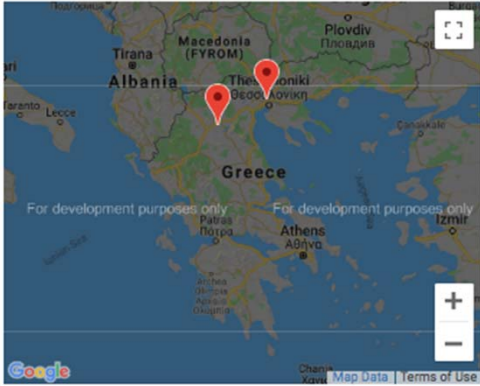
Unique ID:

Track Product

Product Name: Onions

#	Handler Address	Action	Latitude	Longitude	Timestamp
1	0x0564cc7d18be24561d4e17836b28700ad05a21d8	creation	40.736851	22.920227	2018-10-16T13:50:53.000Z
2	0x8d48c65d54f55c8c7f5ab056a7e3bcd126dbab35	reload	40.30069	21.78896	2018-10-18T09:42:58.000Z

Map



Fetching Handler Info

- Handler info can be easily retrieved by invoking the method *getHandler*:

```
function getHandler() {  
    if(web3.isConnected == false) {  
        alert("Get Handler: No connection to the Ethereum")  
    } else {  
        let handadd = $("#handadd").val();  
        contractInstance.getHandler(handadd, function(err1,res1) {  
            let tmpJSON = JSON.parse(web3.toAscii(cleanPadding(res1[1])));  
            ...  
        });  
    }  
}
```


Check connectivity

Clean padding before decoding and parsing

Html document is populated directly

- This method returns the *name* of the Handler as a JSON document encoded in bytes32 and Handler's address as a JSON document encoded in bytes32, based on Handler's Ethereum address. Decoding required invoking `web3.toAscii()`.

Fetching Handler Info


 DECENTRALIZED
TRAINING SERIES

Register HandlerRegister ProductAdd CheckpointTrack ProductGet Handler

Get Handler Info

Handler Address:

[Get Handler](#)

 DECENTRALIZED
TRAINING SERIES

Register HandlerRegister ProductAdd CheckpointTrack ProductGet HandlerList All HandlersList All Products

Get Handler Info

Handler Address:

[Get Handler](#)

Handler Address	Name	Address
0x0564cc7d18be24561d4e17836b28700ad05a21d8	QWER Ltd	Salonika, Greece

Fetching All Handlers

- ▼ Fetching all Handlers is similar to the procedure of Fetching a specific Handler. Invocation of the contract method returns a tuple of arrays.

```
function getAllHandlers() {  
  if(web3.isConnected == false) {  
    alert("Get All Handlers: No connection to the Ethereum")  
  } else {  
    contractInstance.getAllHandlers(function(err1,res1) {  
      let numOfHandlers = res1[0].length;  
      for(let i = 0; i < numOfHandlers; i++) {  
        let tmpJSON = JSON.parse(web3.toAscii(cleanPadding(res1[2][i])));  
        ...  
      }  
      ...  
    });  
  }  
}
```

Check connectivity

Number of Handlers coincides with length of the array of addresses returned.

Custom fields require removal of padding and decoding

- ▼ Tuple of arrays has the form **result[field][entry]**.

Fetching All Handlers

DECENTRALIZED
TRAINING SERIES

Register HandlerRegister ProductAdd CheckpointTrack ProductGet Handler

All Handlers Info

Fetch All Handlers

DECENTRALIZED
TRAINING SERIES

Register HandlerRegister ProductAdd CheckpointTrack ProductGet HandlerList All HandlersList All Products

All Handlers Info

Fetch All Handlers

#	Handler Address	Name	Address
1	0x46acff13bfa9de1ee3fe031867d18c0fe3324b43	ACME Ltd	Xanthi, Greece
2	0xf89cf96e1fd534d022ea9c6138f3ddcb12f75d61	CNME Ltd	Kavala, Greece
3	0x0564cc7d18be24561d4e17836b28700ad05a21d8	QWER Ltd	Salonika, Greece
4	0x8d48c65d54f55c8c7f5ab056a7e3bcd126dbab35	ASDF Ltd	Kozani, Greece

Fetching All Products

- ▼ Fetching all Products include all information as well as Unique IDs for each one. Returning Checkpoints is not suggested for each product, since the amount of data and computational work related increases substantially. Instead the number of Checkpoints is returned.
- ▼ Moreover, avoiding recover of the checkpoints inside the getter in the contract does not encumber the constituent full node. Each product is described by Unique ID, Name and Number of Checkpoints.

```
▼ function getAllProducts() {  
    if(web3.isConnected == false) {  
        alert("Get All Products: No connection to the Ethereum")  
    } else {  
        contractInstance.getAllProducts(function(err1,res1) {  
            let numOfProducts = res1[0].length;  
            for(let i = 0; i < numOfProducts; i++) {  
                let tmpJSON = JSON.parse(web3.toAscii(cleanPadding(res1[1][i])));  
                ...  
            }  
            ...  
        });  
    }  
}
```

Check connectivity

Number of products coincides with the length of the array of Unique IDs

Custom fields require removal of padding and decoding

- ▼ Tuple of arrays has the form **result[field][entry]**.

Fetching All Products

DECENTRALIZED
TRAINING SERIES

Register HandlerRegister ProductAdd CheckpointTrack ProductGet Handler

All Products Info

Fetch All Products

DECENTRALIZED
TRAINING SERIES

Register HandlerRegister ProductAdd CheckpointTrack ProductGet HandlerList All HandlersList All Products

All Products Info

Fetch All Products

#	Unique ID	Name	Number of Checkpoints
1	123	Beetroots	4
2	12	Onions	2

Design Issues

- ▼ All custom input fields resulting to bytes32 encoded JSON data should have prefixed length. A bytes32 string can handle 32 ASCII encoded characters. Due to JSON format input size is limited. For example the action of a Checkpoint is:
`{"action": "<what>"}`
- ▼ This limits **<what>** to $32 - 13 = 19$ characters to describe an action. Similarly all custom fields need special attention when inputting data of arbitrary length. Data that violates length is pruned during conversion to bytes32 and cannot be parsed properly by **JSON.parse()** method leading to errors:
`<input type="text" id="action" maxlength="19" />`
- ▼ The lists retaining all Products and all Handlers could be populated upon load of the webpage, however this would substantially affect experience, since fetching a lot of data from the Blockchain requires substantial amount of time.

Design Issues

- ▼ The presented Distributed Application (DApp) can be used by anyone with a browser supporting MetaMask or any Dapp browser such as Brave browser (<https://brave.com/>) or Trust Wallet's Android DApp Browser or any browser that supports injection of ***web3.currentProvider***.
- ▼ All communications with the Blockchain have been performed through MetaMask and in consequence Infura. Despite the fact that Infura has a scalable infrastructure, several features are not supported. These features include subscription for listening for events. In our case an event is emitted only when an owner changes.
- ▼ Change of Owner has not been implemented in the HTML as a Tab since it is a rarely used procedure and concerns a specific user (owner).

Future Improvements, Suggestions and Discussion

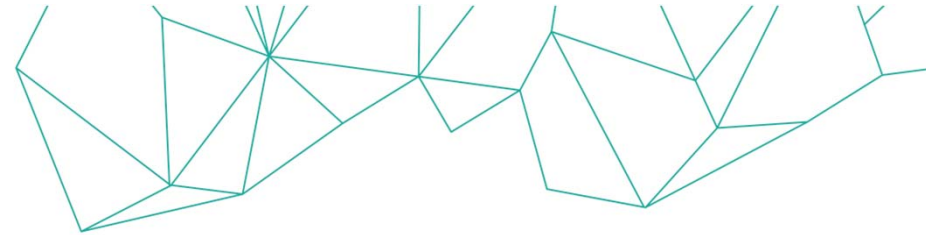
▼ Future Improvements may include:

- ▼ Change of callbacks to promises. Promises are a better and faster way of handling asynchronous operations (Bluebird library).
- ▼ A mechanism to wait on transaction per Handler. When multiple transactions are performed without waiting for confirmation it is highly probable that all of them will be transmitted with the same nonce and the first one accepted leads to rejection of all the others. Another problem is the nonce gap that has to be covered before 50 blocks pass, otherwise the higher nonce transactions are dropped.
- ▼ Addition of events to critical operations such as addition of Handlers.
- ▼ Multilevel user access control. In the current deployment there are only two levels: Owner and Handler that can interact with the information of the contract. However, having a single entity for registering handlers is impractical.
- ▼ Ability to delete – deregister Handlers and Products.
- ▼ Deactivation of tabs based on the role of the user (Handler or Owner).
- ▼ Redesign using a more modern framework such as AngularJS.
- ▼ During deployment of contract optimization should be used.

▼ Any suggestions ?

Guidelines for designing large decentralized applications

- ▼ In large scale applications, it is usual for the backend to be composed of more than the Ethereum Blockchain.
- ▼ In these cases it is preferable to create a server using Python or Node.js and invoke methods through REST calls. In this setting recurring services that track interactions and events can be built.
- ▼ Moreover, caching and local Database storage can be used to reduce computational work related to fetching large amounts of data from the Blockchain. Traversing the Blockchain is computationally expensive.
- ▼ Furthermore, despite the fact that MetaMask is a helpful tool, people usually have different types of wallets. Thus, a local (browser) signing service based on a private key can be also built as a more generic approach.
- ▼ Storage of private keys on nodes should be avoided.



Decentralized Training Series Workshop: Certified Ethereum Specialist

19-20 November 2018, Athens, Greece

Thank you !!! / Questions?

