Security Foundations

Ethereum Lab - Web3 Interface

Nicola Atzei - atzeinicola@unica.it

Setup

Create a new account as shown in the previous lectures. You can charge your account executing the following command

```
wget -q --post-data="address=<address>" -O - http://co2.unica.it/charge-ethereum-account
```

The service in not always available.

First project

Start geth exposing the HTTP API

```
geth --testnet --rpc --rpcapi "web3,eth,personal"
```

Open Webstorm and create a new project **ethereum-lab**

Add the required dependencies

```
sudo apt-get install git
cd $HOME/WebstormProjects/ethereum-lab
npm install web3
```

You should see a new directory **node_modules** within your project

Create a new file **main.js**

Write the following lines

```
var Web3 = require("web3");

var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
var number = web3.eth.blockNumber;

console.log("Block number: "+number);
```

Run your program within Webstorm or on a new terminal

```
nodejs main.js
```

The output should be similar to

```
Block number: 153627
```

Get the balance

The following code query the blockchain to retrieve the balance of a given account, converting the result from wei to ether

```
var account = "0x836b9a551dE259e19a9d28da4feE87fec4254256";
var balanceWei = web3.eth.getBalance(account).toNumber();
var balance = web3.fromWei(balanceWei, 'ether');

console.log("Balance: "+balance);
```

The output should be

```
Balance: 349.2581846720137
```

Calling smart contracts functions

Recall the contract from the previous lesson

```
contract MyContract {
    address public owner;

    function MyContract() {
        owner = msg.sender;
    }

    modifier onlyOwner { if (msg.sender!=owner) throw; _;}
    event paidBy (address from, uint value);

    function changeOwner(address newOwner) onlyOwner {
        owner = newOwner;
    }

    function() payable {
        paidBy(msg.sender, msg.value);
    }

    function withdraw() onlyOwner{
        bool r = owner.send(this.balance);
    }
}
```

The contract was published at address **0x8c55111D20901D62CC2E740003f7034E94e496a2** and its ABI interface is

```
[
      {
         "constant": false, "inputs": [], "name": "withdraw",
         "outputs": [], "payable": false, "type": "function"
      },
      {
         "constant": true, "inputs": [], "name": "owner",
         "outputs": [{ "name": "", "type": "address" }],
         "payable": false, "type": "function"
      },
      {
         "constant": false,
         "inputs": [{
             "name": "newOwner", "type": "address"
         }], "name": "changeOwner",
         "outputs": [],
         "payable": false, "type": "function"
      },
      {
         "inputs": [], "payable": false, "type": "constructor"
      },
      {
         "payable": true, "type": "fallback"
      },
      {
         "anonymous": false,
         "inputs": [
             {"indexed": false, "name": "from", "type": "address" },
             {"indexed": false, "name": "value", "type": "uint256" }
         ],
         "name": "paidBy", "type": "event"
      }
]
```

Instantiate a contract

```
var contractAddr = "0x8c55111D20901D62CC2E740003f7034E94e496a2";
var abiArray = [...];
var contract = web3.eth.contract(abiArray).at(contractAddr);
```

Now you can invoke the *constant* function **owner**

```
var owner = contract.owner();
console.log("The owner is: "+owner);
```

Note: the function is constant. It means that will not be executed with a new transaction, but your node will execute it locally. The result is based on the state of the blockchain at the moment of the invocation. Finally, constant functions cannot modify the state of a contract.

The output should be something like this

```
The owner is: 0x93471f8bc99114ccf0f9c1e84349c9be390b73e8
```

Non constant function can be invoked specifying an object containing the transaction's details. For example, the owner can invoke the function **withdraw** doing (assuming that **account** is the owner)

```
web3.personal.unlockAccount(account, "<password>");          // unlock the account
var tx = contract.withdraw({from: account, gas: 4000000});
console.log("withdraw tx: "+tx);
```

Or alternatively

```
web3.personal.unlockAccount(account, "<password>");
var tx = web3.eth.sendTransaction({
   from: account,
   to: contractAddr,
   data: contract.withdraw.getData(),    // return the function signature
   gas: 4000000
});

console.log("withdraw tx: "+tx);
```

Finally, the fallback function is executed every time the contract is invoked without specifying the data field (e.g. when sending ether to the contract)

```
web3.personal.unlockAccount(account, "<password>");
var tx = web3.eth.sendTransaction({
   from: account,
   to: contractAddr,
   value: web3.toWei(5, "ether")
});

console.log("sending ether with tx: "+tx);
```

Handle an event

Events can be handled, that is you can "subscribe" to an event and execute, for example write the following function

```
function paidByHandler(error, event) {
   if (!error) {
        var contract = event.address;
        var from = event.args.from;
        var value = event.args.value;
        var newBalance = web3.eth.getBalance(contract).toNumber();
        console.log("-> The user "+from+" paid "+value+" wei to "+contract);
        console.log("-> Contract balance is: "+newBalance);
   }
   else
        console.log("-> Error handling the event.")
}
```

Then you can register your contract to execute the function **paidByHandler** every time the contract receives some ether

```
contract.paidBy(paidByHandler);
```

In order to trigger the event, you must send some wei to the contract

```
web3.personal.unlockAccount(account, "<password>");
web3.eth.sendTransaction({
   from: account,
   to: contractAddr,
   value: 1
});

console.log("Transaction sent");
```

After a while, the output should be similar to

```
Transaction sent
-> The user 0x836b9... paid 1 wei to 0x8c551...
-> Contract balance is: 1
```

Wait for a tx to be published (extra)

The API does not provide (so far) a way to send a transaction synchronously. An easy workaround is to pool our node in order to retrieve the transaction receipt, that is the object containing some information related to the transaction's execution. When the receipt is available, the transaction is published. A function **waitTransaction** could be implemented like this

```
function watchTransaction(tx, callback) {
   var filter = web3.eth.filter('latest');     // create a filter to listen for incoming blocks

   filter.watch(function (error, blockhash) {
       if (!error) {
           var block = web3.eth.getBlock(blockhash);   // for each new block, search for the given
transaction
           if (block.transactions.includes(tx)) {
               filter.stopWatching();
               if (callback) callback();
           }
       }
       else {
           console.log("[ERROR] error receiving latest block: "+err)
       }
   });
}
```

The function can be used as follows

```
watchTransaction(tx, function(){
   console.log("mined")
});
```

Proof of presence - 0.5pts

I have published the following contract at address **0x9fcce7ad278991991db0b745a4c65d6480f324f7**.

```
contract ProofOfPresence {

   address public owner;
   function ProofOfPresence() { owner = msg.sender; }

   modifier onlyOwner {
       if (msg.sender!=owner) throw;
       _;
   }

   event tokenEvent (
       uint id,
       uint token
   );

   function triggerToken(uint id, uint token) onlyOwner {
       tokenEvent(id, token);
   }

   struct Log {
       address addr;
       uint id;
       uint token;
   }

   Log[100] public submissions;
   uint tot;

   function submission(uint id, uint token) {
       submissions[tot++] = Log(msg.sender,id,token);
   }
```

```
}
```

I am the owner, so the only possibility you have to get your token is by handling the **tokenEvent**, which is triggered by the invocation of the **triggerToken** function. The function is executed every minute by an application I did.

You have to:

1. determine the ABI interface (you can use the online compiler at Solidity realtime compiler and runtime)
2. create an application to
    a. instantiate a contract by using the provided address and the ABI interface obtained at step 1
    b. listen for the **tokenEvent** and get the token associated to your ID
    c. submit the token by invoking the **submission** function