



Assignment 2 - Smart contracts

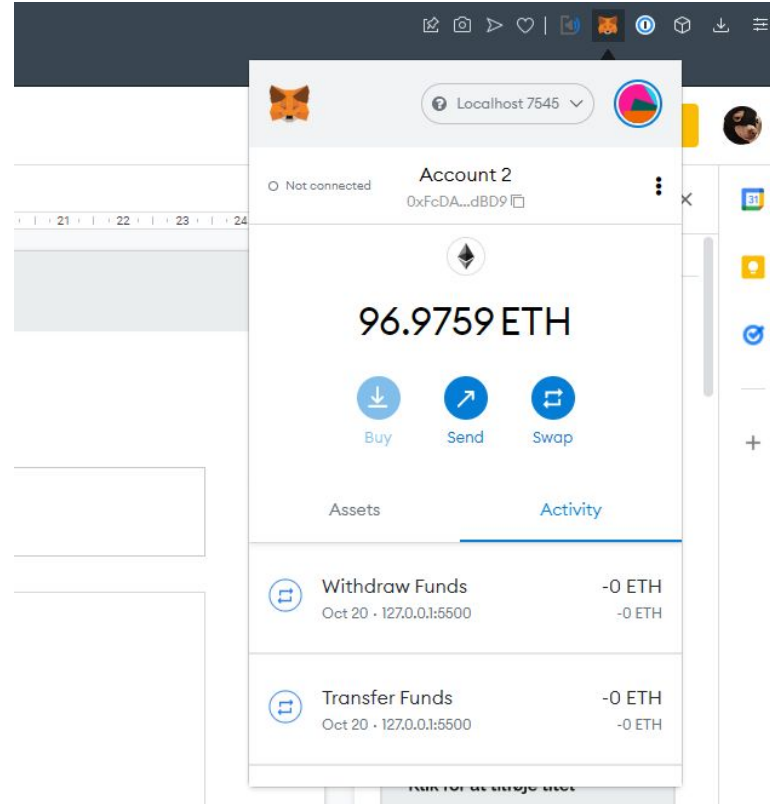
By. Sebastian Christiansen

Getting a crypto wallet

I used **MetaMask** for this.

We simply need a crypto wallet to connect to the website (*more on that later*) to transact - change the state of the contract.

MetaMask needs to connect to the **locally running Ethereum** network...



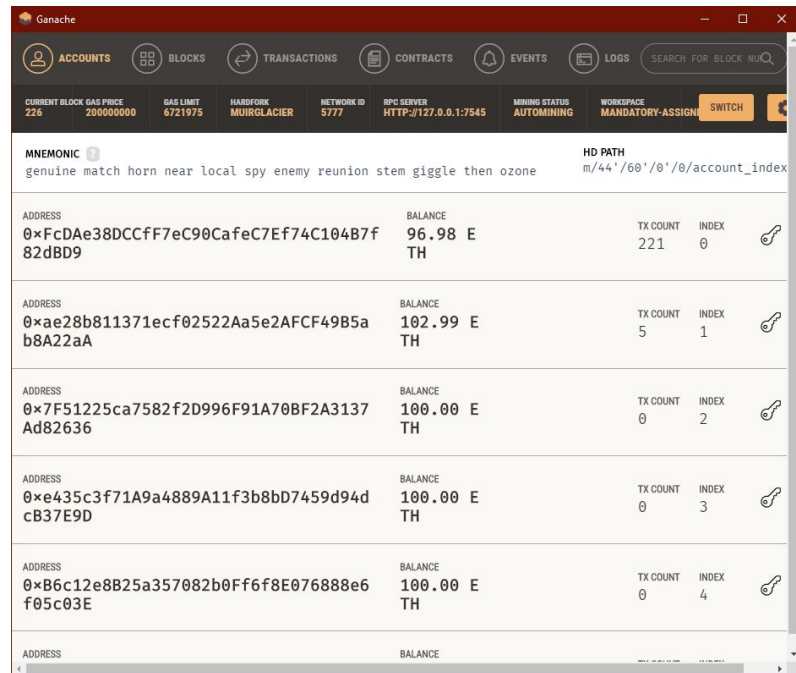
A local Ethereum network

With **Ganache**, one can simply create a new workspace which will spin up a new local **Ethereum network** with its own blockchain.

This is great for **testing** smart contracts as the preparation time is near null.

Ganache also creates 6 (or more) wallets for you to play with, each having 100 ETH by default.

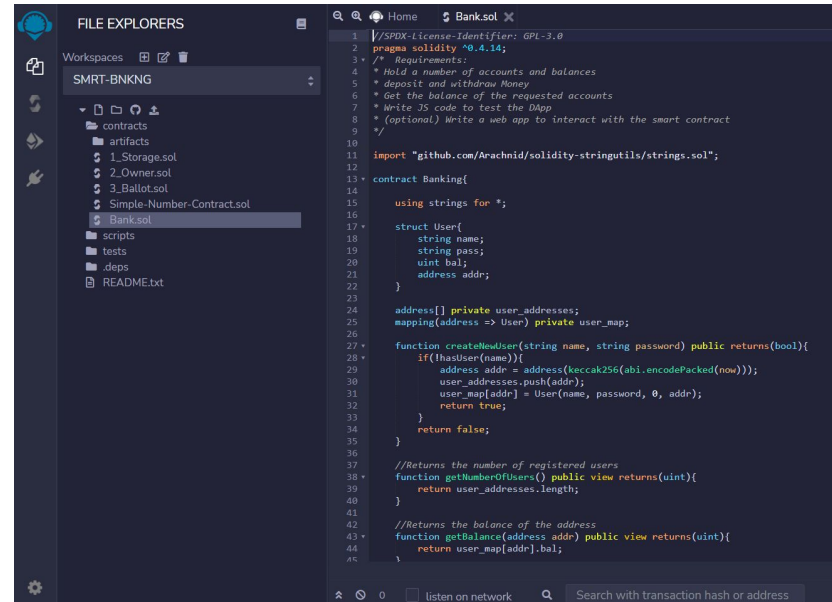
The Ethereum network will be automatically accessible at localhost:7545.



Writing a smart contract

I used Remix (remix.ethereum.org) to write the contract, this includes a compiler (to get an ABI) and a deploy and debug tool.

The language is Solidity



The screenshot displays the Remix IDE interface. On the left, the 'FILE EXPLORERS' panel shows a workspace named 'SMRT-BNKG' with a file tree containing folders for 'contracts', 'artifacts', 'scripts', 'tests', and 'deps', along with a 'README.txt' file. The 'contracts' folder is expanded, showing files like '1_Storage.sol', '2_Owner.sol', '3_Ballot.sol', 'Simple-Number-Contract.sol', and 'Bank.sol'. The main editor on the right displays the code for 'Bank.sol'. The code includes a license header, a pragma statement for Solidity version 0.4.14, a requirements comment, and a contract named 'Banking'. The contract defines a 'User' struct with fields for 'name', 'pass', 'bal', and 'address'. It includes a private array 'user_addresses' and a private map 'user_map'. The contract features three functions: 'createNewUser' which adds a new user to the map and returns a boolean, 'getNumberOfUsers' which returns the length of the 'user_addresses' array, and 'getBalance' which returns the balance of a specific user address.

```
1 //SPDX-License-Identifier: GPL-3.0
2 pragma solidity ^0.4.14;
3 /* Requirements:
4  * Hold a number of accounts and balances
5  * deposit and withdraw Money
6  * Get the balance of the requested accounts
7  * Write JS code to test the DApp
8  * (optional) Write a web app to interact with the smart contract
9  */
10
11 import "github.com/Arachnid/solidity-stringutils/strings.sol";
12
13 contract Banking{
14
15     using strings for *;
16
17     struct User{
18         string name;
19         string pass;
20         uint bal;
21         address addr;
22     }
23
24     address[] private user_addresses;
25     mapping(address => User) private user_map;
26
27     function createNewUser(string name, string password) public returns(bool){
28         if(!hasUser(name)){
29             address addr = address(keccak256(abi.encodePacked(now)));
30             user_addresses.push(addr);
31             user_map[addr] = User(name, password, 0, addr);
32             return true;
33         }
34         return false;
35     }
36
37     //Returns the number of registered users
38     function getNumberOfUsers() public view returns(uint){
39         return user_addresses.length;
40     }
41
42     //Returns the balance of the address
43     function getBalance(address addr) public view returns(uint){
44         return user_map[addr].bal;
45     }
46 }
```



Contract code

Contracts can be thought of as classes, with dependencies, variables, and functions.

Address are simply values in hex.

Mapping these are hashmaps of key => value pairs

Struct objects with specified variables

Function visibility can be public and internal (private)

```
Contract Banking{
    struct User{
        string name;
        string pass;
        uint bal;
        address addr;
    }
    address[] private user_addresses;
    mapping(address => User) private user_map;
    function createNewUser(string name, string password) public returns(bool){
        if(!hasUser(name)){
            address addr = address(keccak256(abi.encodePacked(now)));
            user_addresses.push(addr);
            user_map[addr] = User(name, password, 0, addr);
            return true;
        }
        return false;
    }
}
```



Function return

View: specifies read-only functions

Payable (default, none): specify a change in state, must be compensated with a gas price.

```
function getBalance(address addr) public view returns(uint){  
  
    return user_map[addr].bal;  
}
```



Function multiple return

```
function withdrawFunds(address addr, uint funds) public returns(bool, string){  
    ...  
    return (true, "Withdrew funds");  
    ...  
}
```



Connecting Remix to the Ethereum network

One simply goes to 'deploy and run' and connects it to the 'Web3 provider environment'.

External node request

Note: To use Geth & <https://remix.ethereum.org>, configure it to allow requests from Remix:(see [Geth Docs on rpc server](#))

```
geth --http --http.corsdomain https://remix.ethereum.org
```

To run Remix & a local Geth test node, use this command: (see [Geth Docs on Dev mode](#))

```
geth --http --http.corsdomain="https://remix.ethereum.org" --http.api web3,eth,debug,personal,net --vmdebug --datadir <path/to/local/folder/for/test/chain> --dev console
```

WARNING: It is not safe to use the --http.corsdomain flag with a wildcard: --http.corsdomain *

For more info: [Remix Docs on Web3 Provider](#)

Web3 Provider Endpoint

```
http://127.0.0.1:7545
```

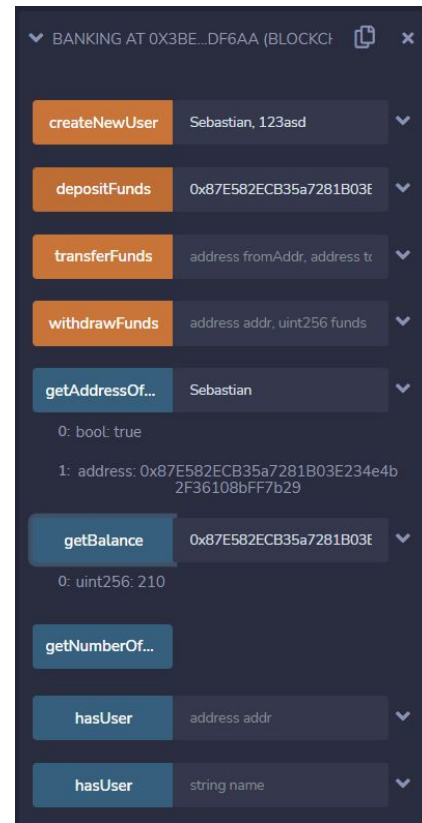
OK

Cancel

Once deployed

Once the smart contract is deployed, we can test in on-site easily.

We can easily call functions of the contract and check the returned values (*easier than website calls*).





Writing a web application

We can instantiate a Web3 object, then send a request to connect accounts from a wallet (MetaMask).

Once the Web3 is instantiated and accounts loaded we can store the contract using its ABI and Address on the blockchain.

The ABI (Application Binary Interface) is used to describe the functions of the Contract.

The webpage requires a JS runtime - I use the 'live server' extension to VS code for this.

```
<script type="text/javascript">
  var web3;
  var accounts;

  var user = null;
  var address = null;

  async function loadWeb3(){
    if(window.ethereum) {
      web3 = await new Web3(window.ethereum);
      await window.ethereum.send('eth_requestAccounts');
      accounts = await window.ethereum.request({ method: 'eth_accounts' });
    }
  }

  async function load(){
    await loadWeb3()
      .then( () => {loadContract()})
  }

  var contractABI = [ ...
  var contractAddr = '0xfa0F528fFc702aEe6740E1afC050cd035c510435'
  function loadContract(){
    window.contract = new web3.eth.Contract(contractABI, contractAddr)
  }

  load();
```



Webpage

Hello, Sebas

Log out

Your current balance is \$30,004.00

Withdraw

Deposit

Making a transfer?

Receipient name

Transfer



Webpage calls to smart contract

```
await window.contract.methods.getBalance(address).call()
```

We can simply access (and await) the contract methods as stored earlier.

The `call()` specifies a read-only operation on the state of the contract. This is free, as it does not change the state.



Webpage transaction to smart contract

```
async function createNewUser(){  
  var name = document.getElementById("createNameField").value;  
  var pass = document.getElementById("createPassField").value;  
  var fromAcc = await web3.eth.getAccounts();  
  await window.contract.methods.createNewUser(name, pass).send({from: fromAcc.toString()})  
  ...  
}
```

Again I access the stored contracts method, and pass the proper parameters.

The send() specifies a transaction, these cost gas, thus I must specify which account to pay with. The payment is only a gas price, used to compensate miners for processing the transaction.



That's it..

The rest is just HTML, which is unimportant for this project.