

# CRUD Smart Contract Manual

Blockchain based smart contract

## Contents

Introduction.....	3
System diagram.....	3
<b>Part One: Setup Environment.....</b>	<b>4</b>
Install the Following Tools/Software.....	6
How to setup Ganache and run the application.....	7
<b>Part Two: Compile Code.....</b>	<b>7</b>
Add Code into the Remix IDE.....	8
Crud.Sol compiled code.....	10
<b>Part three: Connecting to Local Ethereum Network.....</b>	<b>10</b>
To Run Smart Contract on the remix IDE.....	13
To Deploy Contract/transactions onto local Ethereum Virtual Network	14
Critical analysis .....	15

## **Introduction:**

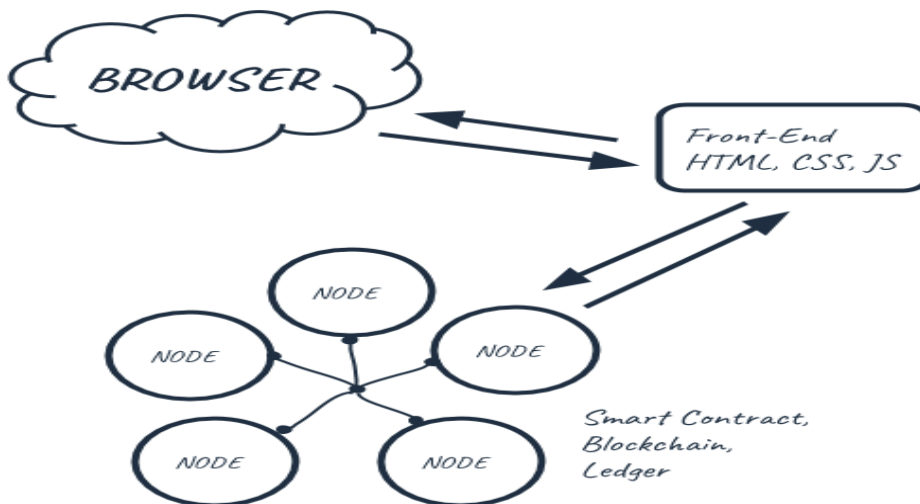
### **What is a blockchain?**

- A good analogy to understand what a blockchain is, is the bitcoin currency. Normally when you try to withdraw from the bank on your computer, you use a browser and talk to a central server over a network. All of the code of this web application lives on the central server, and all the data lives in a centralized database. The centralized database means that there is only one location where the data is stored, which makes the database more prone to attacks and if there is a failure, then the data cannot be retrieved. However, with a blockchain the data is spread out in a chain of blocks which makes the data decentralized. The blocks are also called nodes and are identifiable by a certain hash, generated by a hash algorithm. Blocks on the blockchain also have transaction history, the blockchain is a ledger which records all transactions and keeps up to date info of who owns what and how much they own. The data on the blockchain is immutable and is publicly available to everyone. Meaning, anyone can add to the blockchain because it is a public ledger. In conclusion, The blockchain is a database in itself, and the nodes talking to each other makes it a network.

### **How does the blockchain relate to CRUD?**

- CRUD stands for Create-Read-Update-Delete, which are the basic operations of persistent storage. You cannot delete written transactions on a blockchain, so in this example we are showing you that the transactions cannot be deleted. In blockchain technology, the term we use is CRAB, which stands for Create-Retrieve-Append-Burn. You are burning the code of the smart contract, but the transaction of the variable will always be on the blockchain, so you will always know the past state.

### **System Diagram:**

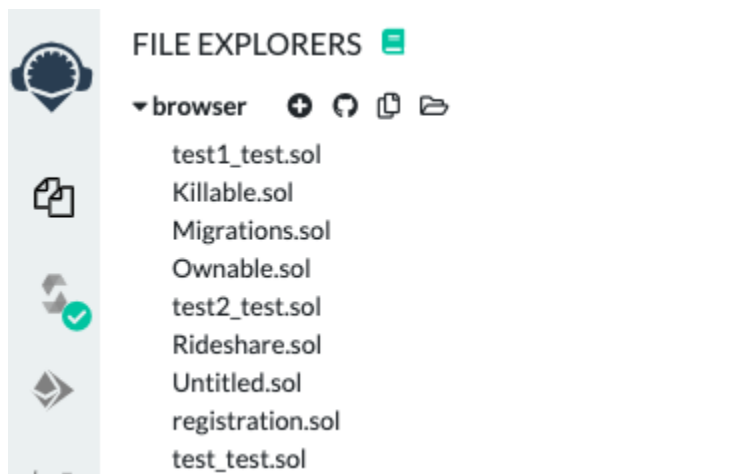


**Fig 1: system diagram for a decentralized application using blockchain technology**

This picture shows how a decentralized application would work. The front-end client would be talking to a network of nodes to store our data instead of a conventional database like Hadoop, SQL, etc. This network of nodes called the blockchain also has smart contracts, and the public ledger of accounts, the smart contracts hold the execution logic for the DAPP. Lastly you use the web browser to interact with the web application and the blockchain network as well through the web3 provider “Ganache”.

### **Simple Smart contract tutorial:**

Before we make the CRUD smart contract, let’s start with a simple storage contract. Go to [remix.ethereum.org](https://remix.ethereum.org).



**Fig 2: the remix IDE user interface for the different files**

Looking at Figure 2, press the small + button that is to the right of the word “browser”. Make a new contract called “SimpleStorage.sol”. and copy and paste the following code (fig.3) into the contract. The code contains one variable called “storedData” and two functions that set the value of the stored data and then returns the value.

```
Pragma solidity >= 0.4.0 < 0.6.0;
contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```


```



}
}


```


**Fig 3: the code for the “SimpleStorage” smart contract**


After copying and pasting the code above, go to the compile tab where you can compile the code. Press on the “compile” button to compile the code, as shown in figure 4. The compiler button is right underneath the file directory button

SOLIDITY COMPILER 

Compiler  0.5.12+commit.7709ece   
 Include nightly builds ☐

Language Solidity 

EVM Version compiler default 

 Compile election1.sol

Compiler Configuration

☒ Auto compile

---



☐ Enable optimization




---

☒ Hide warnings



**Fig 4: the section of the website to compile your code**



After you compile your code you can then run the code, to run the code press on the button underneath the compiler. Refer to Figure 5 to see the transaction screen.

Environment JavaScript VM  

Account  0xCA3...a733c (100 eth)  

Gas limit 3000000

Value 0  wei 

SimpleStorage - browser/SimpleStor  

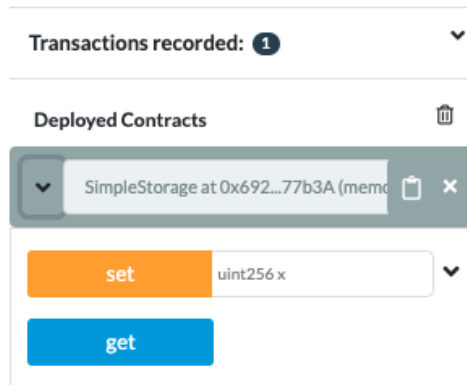
Deploy

or

At Address  Load contract from Address

**Fig 5: the section of the website to run your code**

Once you arrive here, press on “deploy” to deploy your smart contract. The other settings can stay the same for now. After you press deploy look underneath to see the contract is created.



**Fig 6: the section of the website to simulate the simple storage**

Once you are here, type in any value into the box next to the “set” button and then press on the “get” button to return this value. This value is an integer value and behind the scenes is stored in a blockchain, permanently. Go to the right to see all the metrics about the transaction that you just appended to the blockchain.



**Fig 7: Transaction details**

click on the arrow to the right of the “debug” button, this will show the complete details of the transaction that just happened.

## **STEP 1: download the following software below**

### **Set up the environment:**

**Ganache** - Local in memory blockchain for development purposes. Installed at <https://www.trufflesuite.com/ganache>. This application gives you test accounts and private keys to use with Metamask. Each account holds 100 fake ether.

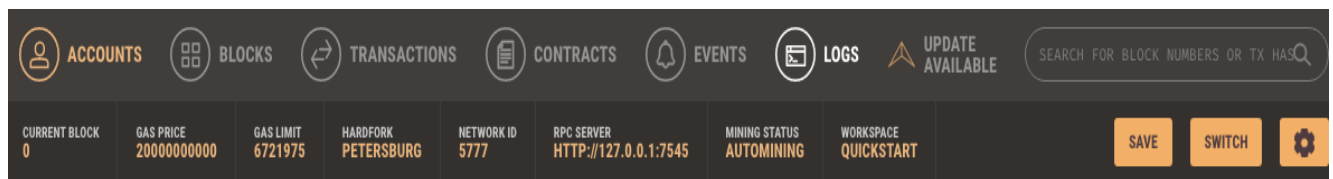
**Remix IDE** – all the development and testing will be done on an online IDE called remix IDE, the website is <https://remix.ethereum.org/>.

**Step 2: run Ganache** The first step is to start the Ganache application, then press “quick start” and it should look like this with your accounts:

ADDRESS 0x8758Ed187224f714946C3A714BfD1a92B2399A5a	BALANCE 99.90 ETH	TX COUNT 28	INDEX 0	
ADDRESS 0x55DCdA75F5293611DAD2bAa135fA59DD5B1dE2bB	BALANCE 100.00 ETH	TX COUNT 6	INDEX 1	
ADDRESS 0x148a1D979F9bF536f1FcBd7829a363DC150C6495	BALANCE 100.00 ETH	TX COUNT 0	INDEX 2	
ADDRESS 0x5869D0835AE3264111FDF6d69fe22F3505315Ea1	BALANCE 100.00 ETH	TX COUNT 0	INDEX 3	
ADDRESS 0xF8bCF2a824274E1caF09a4689898f14A9731d14c	BALANCE 100.00 ETH	TX COUNT 0	INDEX 4	
ADDRESS 0xEe0Dd19Fa5AAcc02dF0399FFBd4DccFF3c6655E6	BALANCE 100.00 ETH	TX COUNT 0	INDEX 5	
ADDRESS 0x69b58525c8aE3B6af098471e9e9692776EeD7bC2	BALANCE 100.00 ETH	TX COUNT 0	INDEX 6	

**Fig 8: The virtual Ganache test network accounts, with fake ether in them.**

make note of the RPC server address, which is the server port and address that you have to use for testing your local network.



**Fig 9: The top of the Ganache application showing the Server ip and port.**

**Step 3: open the Remix IDE and copy and paste the finished code.**

1. Make sure you have the previous software downloaded and make note of your RPC server value, the one in the above example is 127.0.0.1:7545, the Mining status is automatically set to “automining” which means the blocks are automatically mined as a transaction.

### **Coding the smart contract:**

We continue coding the smart contract by writing code for creating, reading, updating and deleting. The user is stored in a struct variable, which contains the name and a true or false whether it has been created or not.

Below is the complete code for the smart contract:

```
pragma solidity >=0.4.0 < 0.6.0;

contract crud {

    struct User {
        string name;
        bool created;
    }

    address[] public users;
    mapping(address=>User) public username;
    uint public userIterator;

    function createUser(string memory _name) public {
        require(!username[msg.sender].created);
        username[msg.sender].name = _name;
        users.push(msg.sender);
        userIterator+=1;
        username[msg.sender].created = true;
    }

    function updateUser(string memory _name) public {
        username[msg.sender].name = _name;
    }

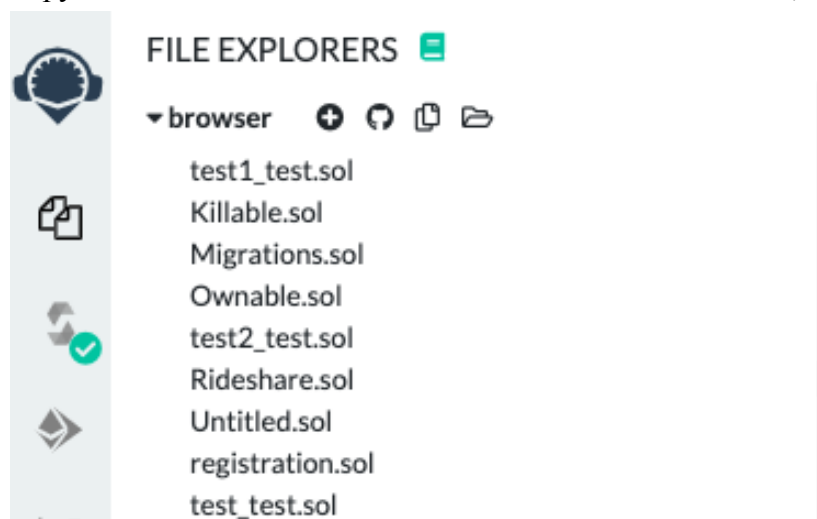
    function readUser(address _userAddress) view public returns (string memory) {
        for(uint i = 0; i < users.length; i++) {
            if(users[i] == _userAddress) {
                return(username[msg.sender].name);
            }
            continue;
        }
    }
}
```



```
function removeUser(address _userAddress) public {
    for(uint i = 0; i < users.length; i++){
        if(users[i] == _userAddress) {
            delete(users[i]);
            delete(username[msg.sender]);
            userIterator--;
        }
    }
}
}
```

**Fig 11: complete code for the Election smart contract**

Copy the code above and head over to the remix IDE website, which the link is provided earlier.



**Fig 12: Remix IDE panel**

On the panel, click on the little plus button and type “crud.sol” for the name of the smart contract, then paste the code you copied above into the smart contract.

The complete code will be explained right now:

- The User struct is made to hold a specific name with a specific address. After the struct, we require a variable to store the users. This is done with a Mapping variable in Solidity. The mapping type variable is an hash which stores a key-value pair. In our example the key is an address, meanwhile the mapping value is a user. We can now look up a specific user with their address.

```
// Store users into the blockchain and read their data, update, and delete
pragma solidity >=0.4.0 < 0.6.0;
```

```
contract crud {

struct User {
    string name;
    bool created;
}

mapping(address=>User) public username;
```

**Fig 12: mapping variable declared in the crud.sol smart contract**

- The next declaration is a variable to hold the number of users
- If we try to look up a user's number higher than the number of users we have stored in our users array, a default value will be returned for the user. We have to use the iterator to figure out how many users there are.

```
address[] public users;
mapping(address=>User) public username;
uint public userIterator;
```

**Fig 13: variables like the unsigned integer variable to hold the count of users are declared**

The function “createUser” takes a value for the name of the user as a parameter. The function also requires that the user's created value is set to false, and then sets the username of the user in the mapping variable to the name passed as the name parameter.

```
function createUser(string memory _name) public {
    require(!username[msg.sender].created);
    username[msg.sender].name = _name;
    users.push(msg.sender);
    userIterator+=1;
    username[msg.sender].created = true;
}
```

**Fig 14: The “createUser” function for the smart contract.**

```
constructor() public {
```

```
}
```

**Fig 15: the “constructor” function is created when the contract is made**

As we can see in figure 15, the “constructor” function must be made to make the smart contract initialized, however it doesn’t need to be initialized to anything, so in this case we left it blank.

Side note: A integer variable was previously created which holds the total count of users that have been created. Also if you haven’t noticed by now, we are specifically making this simulation only allow one account to have one name.

```
function updateUser(string memory _name) public {
    username[msg.sender].name = _name;
}
```

**Fig 16: updateUser function is created to update a user’s name**

The “readUser” function reads in a users address as the parameter and tries to find the address and then returns the value of the “name” variable corresponding to the address of whom called the function.

```
function readUser(address _userAddress) view public returns (string memory) {
    for(uint i = 0; i < users.length; i++) {
        if(users[i] == _userAddress) {
            return(username[msg.sender].name);
        }
        continue;
    }
}
```

**Fig 17: readUser function which takes in a user’s Address**

The last part of the contract corresponds to the last letter of CRUD, which is deletion. The accounts address is then deleted from the “users” array, as well as the “username” mapping variable. This is the first time we encountered a for loop too, which is a way of attacking an iterator to an object and checking each entry one by one of the array.

```
function removeUser(address _userAddress) public {
```


```


for(uint i = 0; i < users.length; i++){
    if(users[i] == _userAddress) {
        delete(users[i]);
        delete(username[msg.sender]);
        userIterator--;
    }
}
}
}


```

**Fig 18:removeUser function which takes in a user's address**

### **Step 3: Testing the code on Remix IDE**


SOLIDITY COMPILER 

Compiler 


0.5.12+commit.7709ece 


Include nightly builds ☐

Language

Solidity 

EVM Version

compiler default 

 Compile election1.sol

Compiler Configuration

☒ Auto compile

---

☐ Enable optimization

---

☒ Hide warnings

---

**Fig 19: The Solidity compiler on the Remix IDE website**

Head to the compiler section and then compile your smart contract, by pressing on the button which is in Figure 19. After the smart contract is compiled, head over to “deploy and run transactions” section which is right underneath and will allow you to run your code.

**DEPLOY & RUN TRANSACTIONS**

Environment: Web3 Provider Custom (5777) network

Account: 0x476...aec6d (99.9781%) Copy Edit

Gas limit: 3000000

Value: 0 wei

election - browser/election1.sol

**Fig 20: section where you can deploy your smart contract and record a transaction**

Next to deploy the actual smart contract, first change the Environment to “Web3 Provider” this will connect your local blockchain “Ganache” to the remix IDE. You have to type in the port for the Ganache network.

Environment: Web3 Provider Custom (5777) network

Account: 0x8c9...4f38d (99.782%) Copy Edit

Gas limit: 3000000

Value: 0 wei

crud - browser/crud1.sol


**Deploy**


or



**At Address** Load contract from Address

**Fig 21: button to deploy your smart contract to the blockchain**

The smart contract is now deployed as a transaction the blockchain. Next head over to the section which is in figure 22.

DEPLOY & RUN TRANSACTIONS 

Deployed Contracts 

▼ crud at 0x3Fd...AB81B (blockchain)  

**createUser** "mark" ▼

**removeUser** 0xee8e5e1428a4a5f0de0c44 ▼

**updateUser** "chad" ▼

**howManyUsers**

0: uint256: 1

---

**readUser** 0x8c91ff851f4df759250769! ▼

0: string: chad

---

**userIterator**

0: uint256: 1


---


**username** 0x8c91ff851f4df759250769! ▼

0: string: name chad




**Fig 22: Deployed smart contract interface**

This is where we can actually test our smart contract and simulate it. Make sure to always change the account address that you use with the drop down menu above. These account address coordinate with your Ganache account addresses and are required to make a new user. Also copy the account address by pressing the clipboard button.


DEPLOY & RUN TRANSACTIONS 



Environment Web3 Provider 

Custom (5777) network

Account  0x476...aec6d (99.9781)  

Gas limit 3000000

Value 0 wei 

election - browser/election1.sol  

**Fig 23: deploy and run transactions page with the specific account address.**

The orange buttons are functions that write data to the blockchain and modify data, while the blue buttons are smart contract functions that simply return data to the user.. Keep playing around with the smart contract and Remix IDE interface's other functions and try and guess what they will do from the smart contract code provided.

## **Critical Analysis**

Learning Outcomes:

The knowledge and skills that we used in the project that we acquired in earlier courses include:

**Programming:**

- Solidity

**Other:**

- Security (Hashing)
- Terminal commands
- Software Design concepts

**The new knowledge and skills we acquired from this project include:**

- Smart contracts, Blockchain, Solidity language syntax, Ethereum networks understanding,