```
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!

Commands:

  Compile:        truffle compile
  Migrate:        truffle migrate
  Test contracts: truffle test
```

Dapp/ Smart Contract - from Beginning to End:

HashMarket:

Get global truffle version with constructor support

```
npm install -g truffle@4.1.7
```

Mkdir-hashmarket

Cd to it

Setting up Truffle

Truffle init

You get this

```
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!

Commands:

  Compile:        truffle compile
  Migrate:        truffle migrate
  Test contracts: truffle test
```

Your folder structure should look like this:

Install ganache global (if you haven’t)

Run ganache-cli or GUI

ganach-cli

get accounts- make sure that you see the accounts with Ether

Next

After this, go to your `truffle.js` or `truffle-config.js` file and add a development network to your config:

```
module.exports = {
    networks: {
      development: {
        host: "127.0.0.1",
        port: 8545,
        network_id: "*"
      }
    }
};
```

Writing the Smart Contract:

The first thing we'll do is write the HashMarket smart contract. We'll try to keep it as simple as possible, while still retaining the required functionality.

HashMarket is a type of eBay on the blockchain. It enables sellers to post products and buyers to buy them for ether. It also enables sellers to remove products if they're not sold.

In your project, in the `contracts` folder, create a new file and call it `HashMarket.sol`. In that file, add the following code:

```
pragma solidity 0.4.21;


contract HashMarket {

    // Track the state of the items, while preserving history
    enum ItemStatus {
        active,
        sold,
        removed
    }

    struct Item {
        bytes32 name;
        uint price;
        address seller;
        ItemStatus status;
    }

    event ItemAdded(bytes32 name, uint price, address seller);
    event ItemPurchased(uint itemID, address buyer, address
seller);
    event ItemRemoved(uint itemID);
    event FundsPulled(address owner, uint amount);
```

```
    Item[] private _items;
    mapping (address => uint) public _pendingWithdrawals;

    modifier onlyIfItemExists(uint itemID) {
        require(_items[itemID].seller != address(0));
        _;
    }

    function addNewItem(bytes32 name, uint price) public returns
(uint) {

        _items.push(Item({
            name: name,
            price: price,
            seller: msg.sender,
            status: ItemStatus.active
        }));

        emit ItemAdded(name, price, msg.sender);
        // Item is pushed to the end, so the lenth is used for
        // the ID of the item
        return _items.length - 1;
    }

    function getItem(uint itemID) public view
onlyIfItemExists(itemID)
    returns (bytes32, uint, address, uint) {

        Item storage item = _items[itemID];
        return (item.name, item.price, item.seller,
uint(item.status));
    }

    function buyItem(uint itemID) public payable
onlyIfItemExists(itemID) {

        Item storage currentItem = _items[itemID];

        require(currentItem.status == ItemStatus.active);
        require(currentItem.price == msg.value);

        _pendingWithdrawals[currentItem.seller] = msg.value;
        currentItem.status = ItemStatus.sold;

        emit ItemPurchased(itemID, msg.sender,
currentItem.seller);
```

```
    }

    function removeItem(uint itemID) public
onlyIfItemExists(itemID) {
        Item storage currentItem = _items[itemID];

        require(currentItem.seller == msg.sender);
        require(currentItem.status == ItemStatus.active);

        currentItem.status = ItemStatus.removed;

        emit ItemRemoved(itemID);
    }

    function pullFunds() public returns (bool) {
        require(_pendingWithdrawals[msg.sender] > 0);

        uint outstandingFundsAmount =
_pendingWithdrawals[msg.sender];

        if (msg.sender.send(outstandingFundsAmount)) {
            emit FundsPulled(msg.sender,
outstandingFundsAmount);
            return true;
        } else {
            return false;
        }
    }
}
```

After writing or copying code.. always go to remix and check the code for errors:

remix.ethereum.org

You can get the latest code from the repository

https://github.com/BlockVisors/StudentSmartContracts

Compiling and Migrating:

After you've done this, try running `truffle compile` to see if the code will compile
Do this from the project root directory: HashMarket

Truffle compile…

This creates the json abi file in the build/ folder

Then truffle migrate is our way to deploy—

Writing the migration

You need to write a migration to let Truffle know how to deploy your contract to the blockchain. Go into the `migrations` folder and create a new file called `2_deploy_contracts.js`. In that file, add the following code:

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

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

Now do the migration:

Now run `truffle migrate` and hopefully you'll get something like this:

```
Using network 'development'.


Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0xad501b7c4e183459c4ee3fee58ea9309a01aa345f053d053b7a9d168e6efaeff
  Migrations: 0x9d69f4390c8bb260eadb7992d5a3efc8d03e157e
Saving successful migration to network...
  ... 0x7deb2c3d9dacd6d7c3dc45dc5b1c6a534a2104bfd17a1e5a93ce9aade147b86e
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying HashMarket...
  ... 0xcbc967b5292f03af2130fc0f5aaced7080c4851867abd917d6f0d52f1072d91e
  HashMarket: 0x7918eaef5e6a21a26dc95fc95ce9550e98e789d4
Saving successful migration to network...
  ... 0x5b6a332306f739b27ccbdfd10d11e60200b70a55cc775e7165358b711082cf55
Saving artifacts...
```

Solidity testing

In order to start testing, in the `test` folder in your project, create a file called `TestHashMarket.sol`. The Truffle suite provides us with helper functions for testing, so we need to import those. At the beginning of the file, add:

Now lets set up the test:

```
pragma solidity ^0.4.20;


import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/HashMarket.sol";
```

The first two imports are the important ones.

The `Assert` import gives us access to various testing functions, like `Assert.equals()`, `Assert.greaterThan()`, etc. In this manner, Assert works with Truffle to automate most "boring" code writing.

The `DeployedAddresses` import manages contract addresses for us. Since every time you change your contract, you must redeploy it to a new address *and* even if you don't change it, every time you test the contract should be redeployed to start from pristine state. The `DeployedAddresses` library manages this for us.

Now let add the test code below imports:

```
contract TestHashMarket {

    function testAddingNewProduct() public {
```

```
        // DeployedAddresses.HashMarket() handles contract
address
        // management for us
        HashMarket market =
HashMarket(DeployedAddresses.HashMarket());

        bytes32 expectedName = "T";
        uint expectedPrice = 1000;

        uint itemID = market.addNewItem(expectedName,
expectedPrice);

        bytes32 name;
        uint price;
        address seller;
        uint status;

        (name, price, seller, status) = market.getItem(itemID);

        Assert.equal(name, expectedName, "Item name should
match");
        Assert.equal(price, expectedPrice, "Item price should
match");
        Assert.equal(status, uint(HashMarket.ItemStatus.active),
"Item status at creation should be .active");
        Assert.equal(seller, this, "The function caller should
be the seller");
    }

}
```

Let's look at some of the important parts of a test. Firstly:

`HashMarket market = HashMarket(DeployedAddresses.HashMarket());`
This code uses the `DeployedAddresses` library to create a new instance of the `HashMarket` contract for testing.

`Assert.equal(<current>, <expected>, <message>)`
This part of the code takes care of checking whether two values are equal. If yes, it communicates a success message to the test suite. If not, it communicates a failure. It also appends the message so you can know *where* your code failed.

Go into the smart contract and change the AddNew Item function bottom like this:

```
   // Item is pushed to the end, so the lenth is used for
        // the ID of the item
        return _items.length;
}
```

Now let's run this:

```
truffle test
```

It should fail:

Go back and change to this:

```
   // Item is pushed to the end, so the lenth is used for
        // the ID of the item
        return _items.length - 1;
}
```

Run again:

```
truffle test
```

You should get a much happier message:

```
TestHashMarket
  ✓ testAddingNewProduct (130ms)


1 passing (729ms)
```

You should get passing

Remember:

Change the return statement in AddNewitem to get a fail

Next - we can test in javascript

Fill out the test and run

Truffle test again

Like below:

Truffle enables us to use JavaScript for testing, leveraging the Mocha testing framework. This enables you to write more complex tests and get more functionality out of your testing framework.

Okay, let's write the test. First, in the `test` folder, create a file and call it `hashmarket.js`.

The first thing we need to do, is get the reference to our contract in JavaScript. For that, we'll use Truffle's `artifacts.require(...)` function:

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

Now that we have the reference to the contract, let's start writing tests. To start, we'll use the `contract` function provided to us:

```
contract("HashMarket", function(accounts) {

});
```

This creates a test suite for our contract. Now for testing, we use Mocha `it` syntax:

```
contract("HashMarket", function(accounts) {
    it("should add a new product", function() {

    });
});
```

Our entire hashmarket.js should look like below:

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

contract("HashMarket", function(accounts) {
    it("should add a new product", function() {

        // Set the names of test data
        var itemName = "TestItem";
        var itemPrice = 1000;
        var itemSeller = accounts[0];

        // Since all of our testing functions are async, we
store the
        // contract instance at a higher level to enable access
from
        // all functions
        var hashMarketContract;
```

```
        // Item ID will be provided asynchronously so we extract
it
        var itemID;

        return HashMarket.deployed().then(function(instance) {
            // set contract instance into a variable
            hashMarketContract = instance;

            // Subscribe to a Solidity event
            instance.ItemAdded({}).watch((error, result) => {
                if (error) {
                    console.log(error);
                }
                // Once the event is triggered, store the result
in the
                // external variable
                itemID = result.args.itemID;
            });

            // Call the addNewItem function and return the
promise
            return instance.addNewItem(itemName, itemPrice,
{from: itemSeller});
        }).then(function() {
            // This function is triggered after the addNewItem
call transaction
            // has been mined. Now call the getItem function
with the itemID
            // we received from the event
            return hashMarketContract.getItem.call(itemID);
        }).then(function(result) {
            // The result of getItem is a tuple, we can
deconstruct it
            // to variables like this
            var [name, price, seller, status] = result;

            // Start testing. Use web3.toAscii() to convert the
result of
            // the smart contract from Solidity bytecode to
ASCII. After that
            // use the .replace() to pad the excess bytes from
bytes32
            assert.equal(itemName, web3.toAscii(name).replace(/
\u0000/g, ''), "Name wasn't properly added");
            // Use assert.equal() to check all the variables
```

```
            assert.equal(itemPrice, price, "Price wasn't
properly added");
            assert.equal(itemSeller, seller, "Seller wasn't
properly added");
            assert.equal(status, 0, "Status wasn't properly
added");
        });
    });
});
```

Run `truffle test` and you should get something like this:

```
TestHashMarket
  ✓ testAddingNewProduct (109ms)


Contract: HashMarket
  ✓ should add a new product (64ms)



2 passing (876ms)
```

We can extend the test by writing tests for the other functions

Now let build our web interface

Reference:

https://www.sitepoint.com/truffle-testing-smart-contracts/