pragma solidity ^0.4.2;

contract DappToken {

string public name = "DApp Token";

string public symbol = "DAPP";

string public standard = "DApp Token v1.0";

uint256 public totalSupply;

event Transfer(

address indexed \_from,

address indexed \_to,

uint256 \_value

);

event Approval(

address indexed \_owner,

address indexed \_spender,

uint256 \_value

);

mapping(address => uint256) public balanceOf;

mapping(address => mapping(address => uint256)) public allowance;

function DappToken (uint256 \_initialSupply) public {

balanceOf[msg.sender] = \_initialSupply;

totalSupply = \_initialSupply;

}

function transfer(address \_to, uint256 \_value) public returns (bool success) {

require(balanceOf[msg.sender] >= \_value);

balanceOf[msg.sender] -= \_value;

balanceOf[\_to] += \_value;

Transfer(msg.sender, \_to, \_value);

return true;

}

function approve(address \_spender, uint256 \_value) public returns (bool success) {

allowance[msg.sender][\_spender] = \_value;

Approval(msg.sender, \_spender, \_value);

return true;

}

function transferFrom(address \_from, address \_to, uint256 \_value) public returns (bool success) {

require(\_value <= balanceOf[\_from]);

require(\_value <= allowance[\_from][msg.sender]);

balanceOf[\_from] -= \_value;

balanceOf[\_to] += \_value;

allowance[\_from][msg.sender] -= \_value;

Transfer(\_from, \_to, \_value);

return true;

}

}

Let's take a look at what this smart contract does, and how it implements the ERC-20 standard:

It stores the token name string public name = "DApp Token".

It stores the token symbol for cryptocurrency exchanges string public symbol = "DAPP".

It stores the total supply of tokens in existence uint256 public totalSupply.

It uses a Solidity mapping to store the balance of each account that owns tokens mapping(address => uint256) public balanceOf.

It implements a transfer function to allow users to send tokens to another account.

It implements an approve function that allows another account to spend tokens, like on a cryptocurrency exchange. This updates the allowance mapping to see how much the account is allowed to spend.

It implements a transferFrom that allows another account to transfer tokens.

Watch me build out this smart contract step-by-step with this video.

You can also read through the tests for this smart contract to discover more about how it works. These tests ensure that this smart contract behaves the way we expect. Here is a complete test suite that checks all the behavior for the smart contract:

var DappToken = artifacts.require("./DappToken.sol");

contract('DappToken', function(accounts) {

var tokenInstance;

it('initializes the contract with the correct values', function() {

return DappToken.deployed().then(function(instance) {

tokenInstance = instance;

return tokenInstance.name();

}).then(function(name) {

assert.equal(name, 'DApp Token', 'has the correct name');

return tokenInstance.symbol();

}).then(function(symbol) {

assert.equal(symbol, 'DAPP', 'has the correct symbol');

return tokenInstance.standard();

}).then(function(standard) {

assert.equal(standard, 'DApp Token v1.0', 'has the correct standard');

});

})

it('allocates the initial supply upon deployment', function() {

return DappToken.deployed().then(function(instance) {

tokenInstance = instance;

return tokenInstance.totalSupply();

}).then(function(totalSupply) {

assert.equal(totalSupply.toNumber(), 1000000, 'sets the total supply to 1,000,000');

return tokenInstance.balanceOf(accounts[0]);

}).then(function(adminBalance) {

assert.equal(adminBalance.toNumber(), 1000000, 'it allocates the initial supply to the admin account');

});

});

it('transfers token ownership', function() {

return DappToken.deployed().then(function(instance) {

tokenInstance = instance;

// Test `require` statement first by transferring something larger than the sender's balance

return tokenInstance.transfer.call(accounts[1], 99999999999999999999999);

}).then(assert.fail).catch(function(error) {

assert(error.message.indexOf('revert') >= 0, 'error message must contain revert');

return tokenInstance.transfer.call(accounts[1], 250000, { from: accounts[0] });

}).then(function(success) {

assert.equal(success, true, 'it returns true');

return tokenInstance.transfer(accounts[1], 250000, { from: accounts[0] });

}).then(function(receipt) {

assert.equal(receipt.logs.length, 1, 'triggers one event');

assert.equal(receipt.logs[0].event, 'Transfer', 'should be the "Transfer" event');

assert.equal(receipt.logs[0].args.\_from, accounts[0], 'logs the account the tokens are transferred from');

assert.equal(receipt.logs[0].args.\_to, accounts[1], 'logs the account the tokens are transferred to');

assert.equal(receipt.logs[0].args.\_value, 250000, 'logs the transfer amount');

return tokenInstance.balanceOf(accounts[1]);

}).then(function(balance) {

assert.equal(balance.toNumber(), 250000, 'adds the amount to the receiving account');

return tokenInstance.balanceOf(accounts[0]);

}).then(function(balance) {

assert.equal(balance.toNumber(), 750000, 'deducts the amount from the sending account');

});

});

it('approves tokens for delegated transfer', function() {

return DappToken.deployed().then(function(instance) {

tokenInstance = instance;

return tokenInstance.approve.call(accounts[1], 100);

}).then(function(success) {

assert.equal(success, true, 'it returns true');

return tokenInstance.approve(accounts[1], 100, { from: accounts[0] });

}).then(function(receipt) {

assert.equal(receipt.logs.length, 1, 'triggers one event');

assert.equal(receipt.logs[0].event, 'Approval', 'should be the "Approval" event');

assert.equal(receipt.logs[0].args.\_owner, accounts[0], 'logs the account the tokens are authorized by');

assert.equal(receipt.logs[0].args.\_spender, accounts[1], 'logs the account the tokens are authorized to');

assert.equal(receipt.logs[0].args.\_value, 100, 'logs the transfer amount');

return tokenInstance.allowance(accounts[0], accounts[1]);

}).then(function(allowance) {

assert.equal(allowance.toNumber(), 100, 'stores the allowance for delegated trasnfer');

});

});

it('handles delegated token transfers', function() {

return DappToken.deployed().then(function(instance) {

tokenInstance = instance;

fromAccount = accounts[2];

toAccount = accounts[3];

spendingAccount = accounts[4];

// Transfer some tokens to fromAccount

return tokenInstance.transfer(fromAccount, 100, { from: accounts[0] });

}).then(function(receipt) {

// Approve spendingAccount to spend 10 tokens form fromAccount

return tokenInstance.approve(spendingAccount, 10, { from: fromAccount });

}).then(function(receipt) {

// Try transferring something larger than the sender's balance

return tokenInstance.transferFrom(fromAccount, toAccount, 9999, { from: spendingAccount });

}).then(assert.fail).catch(function(error) {

assert(error.message.indexOf('revert') >= 0, 'cannot transfer value larger than balance');

// Try transferring something larger than the approved amount

return tokenInstance.transferFrom(fromAccount, toAccount, 20, { from: spendingAccount });

}).then(assert.fail).catch(function(error) {

assert(error.message.indexOf('revert') >= 0, 'cannot transfer value larger than approved amount');

return tokenInstance.transferFrom.call(fromAccount, toAccount, 10, { from: spendingAccount });

}).then(function(success) {

assert.equal(success, true);

return tokenInstance.transferFrom(fromAccount, toAccount, 10, { from: spendingAccount });

}).then(function(receipt) {

assert.equal(receipt.logs.length, 1, 'triggers one event');

assert.equal(receipt.logs[0].event, 'Transfer', 'should be the "Transfer" event');

assert.equal(receipt.logs[0].args.\_from, fromAccount, 'logs the account the tokens are transferred from');

assert.equal(receipt.logs[0].args.\_to, toAccount, 'logs the account the tokens are transferred to');

assert.equal(receipt.logs[0].args.\_value, 10, 'logs the transfer amount');

return tokenInstance.balanceOf(fromAccount);

}).then(function(balance) {

assert.equal(balance.toNumber(), 90, 'deducts the amount from the sending account');

return tokenInstance.balanceOf(toAccount);

}).then(function(balance) {

assert.equal(balance.toNumber(), 10, 'adds the amount from the receiving account');

return tokenInstance.allowance(fromAccount, spendingAccount);

}).then(function(allowance) {

assert.equal(allowance.toNumber(), 0, 'deducts the amount from the allowance');

});

});

});

You can run the test from the command line with truffle like this:

$ truffle test

Crowd Sale Smart Contract

The accompanying video footage for this portion of the tutorial begins with this video. You can download the code for this smart contract from from github here.

Now we can build a crowd sale smart contract that will allow investors to purchase tokens in an initial coin offering (ICO). Here is the complete crowd sale smart contract Solidity code:

pragma solidity ^0.4.2;

import "./DappToken.sol";

contract DappTokenSale {

address admin;

DappToken public tokenContract;

uint256 public tokenPrice;

uint256 public tokensSold;

event Sell(address \_buyer, uint256 \_amount);

function DappTokenSale(DappToken \_tokenContract, uint256 \_tokenPrice) public {

admin = msg.sender;

tokenContract = \_tokenContract;

tokenPrice = \_tokenPrice;

}

function multiply(uint x, uint y) internal pure returns (uint z) {

require(y == 0 || (z = x \* y) / y == x);

}

function buyTokens(uint256 \_numberOfTokens) public payable {

require(msg.value == multiply(\_numberOfTokens, tokenPrice));

require(tokenContract.balanceOf(this) >= \_numberOfTokens);

require(tokenContract.transfer(msg.sender, \_numberOfTokens));

tokensSold += \_numberOfTokens;

Sell(msg.sender, \_numberOfTokens);

}

function endSale() public {

require(msg.sender == admin);

require(tokenContract.transfer(admin, tokenContract.balanceOf(this)));

// Just transfer the balance to the admin

admin.transfer(address(this).balance);

}

}

Let's take a look at what this smart contract does, and how it functions as a crowd sale:

It stores an admin account for the crowd sale address admin.

It references the ERC-20 token smart contract DappToken public tokenContract.

It stores the token price uint256 public tokenPrice.

It stores the number of tokens sold uint256 public tokensSold.

It implements a sell event so that consumers can get notifications whenever a token has been sold.

It implements a buyTokens function that allows users to purchase tokens in the crowd sale.

It implements an endSale function that allows an admin to end the crowd sale and collect the Ether funds that was raised during the sale.

Watch me build out this smart contract step-by-step with this video.

You can also read through the tests for this smart contract to discover more about how it works. These tests ensure that this smart contract behaves the way we expect. Here is a complete test suite that checks all the behavior for the smart contract:

var DappToken = artifacts.require('./DappToken.sol');

var DappTokenSale = artifacts.require('./DappTokenSale.sol');

contract('DappTokenSale', function(accounts) {

var tokenInstance;

var tokenSaleInstance;

var admin = accounts[0];

var buyer = accounts[1];

var tokenPrice = 1000000000000000; // in wei

var tokensAvailable = 750000;

var numberOfTokens;

it('initializes the contract with the correct values', function() {

return DappTokenSale.deployed().then(function(instance) {

tokenSaleInstance = instance;

return tokenSaleInstance.address

}).then(function(address) {

assert.notEqual(address, 0x0, 'has contract address');

return tokenSaleInstance.tokenContract();

}).then(function(address) {

assert.notEqual(address, 0x0, 'has token contract address');

return tokenSaleInstance.tokenPrice();

}).then(function(price) {

assert.equal(price, tokenPrice, 'token price is correct');

});

});

it('facilitates token buying', function() {

return DappToken.deployed().then(function(instance) {

// Grab token instance first

tokenInstance = instance;

return DappTokenSale.deployed();

}).then(function(instance) {

// Then grab token sale instance

tokenSaleInstance = instance;

// Provision 75% of all tokens to the token sale

return tokenInstance.transfer(tokenSaleInstance.address, tokensAvailable, { from: admin })

}).then(function(receipt) {

numberOfTokens = 10;

return tokenSaleInstance.buyTokens(numberOfTokens, { from: buyer, value: numberOfTokens \* tokenPrice })

}).then(function(receipt) {

assert.equal(receipt.logs.length, 1, 'triggers one event');

assert.equal(receipt.logs[0].event, 'Sell', 'should be the "Sell" event');

assert.equal(receipt.logs[0].args.\_buyer, buyer, 'logs the account that purchased the tokens');

assert.equal(receipt.logs[0].args.\_amount, numberOfTokens, 'logs the number of tokens purchased');

return tokenSaleInstance.tokensSold();

}).then(function(amount) {

assert.equal(amount.toNumber(), numberOfTokens, 'increments the number of tokens sold');

return tokenInstance.balanceOf(buyer);

}).then(function(balance) {

assert.equal(balance.toNumber(), numberOfTokens);

return tokenInstance.balanceOf(tokenSaleInstance.address);

}).then(function(balance) {

assert.equal(balance.toNumber(), tokensAvailable - numberOfTokens);

// Try to buy tokens different from the ether value

return tokenSaleInstance.buyTokens(numberOfTokens, { from: buyer, value: 1 });

}).then(assert.fail).catch(function(error) {

assert(error.message.indexOf('revert') >= 0, 'msg.value must equal number of tokens in wei');

return tokenSaleInstance.buyTokens(800000, { from: buyer, value: numberOfTokens \* tokenPrice })

}).then(assert.fail).catch(function(error) {

assert(error.message.indexOf('revert') >= 0, 'cannot purchase more tokens than available');

});

});

it('ends token sale', function() {

return DappToken.deployed().then(function(instance) {

// Grab token instance first

tokenInstance = instance;

return DappTokenSale.deployed();

}).then(function(instance) {

// Then grab token sale instance

tokenSaleInstance = instance;

// Try to end sale from account other than the admin

return tokenSaleInstance.endSale({ from: buyer });

}).then(assert.fail).catch(function(error) {

assert(error.message.indexOf('revert' >= 0, 'must be admin to end sale'));

// End sale as admin

return tokenSaleInstance.endSale({ from: admin });

}).then(function(receipt) {

return tokenInstance.balanceOf(admin);

}).then(function(balance) {

assert.equal(balance.toNumber(), 999990, 'returns all unsold dapp tokens to admin');

// Check that the contract has no balance

balance = web3.eth.getBalance(tokenSaleInstance.address)

assert.equal(balance.toNumber(), 0);

});

});

});