Lecture 02: Baby's First Smart Contract

Nick Zoghb

BLOCKCHAIN
AT BERKELEY

LAB OUTLINE

1. TOOLS OF THE TRADE
2. MORE SOLIDITY
3. EXAMPLE
4. EXERCISE: SOLIDITY + TRUFFLE

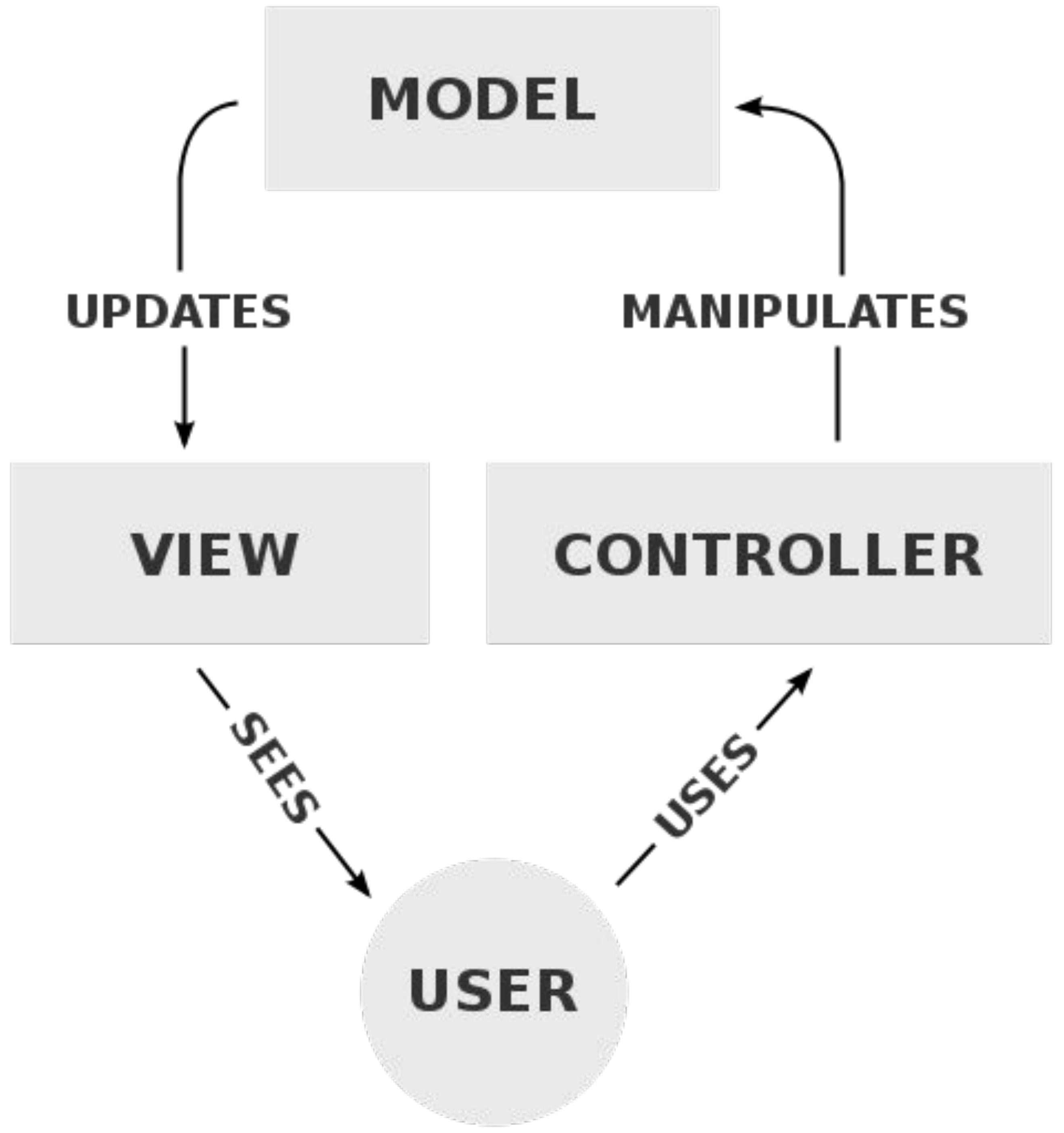
1

TOOLS OF THE TRADE

ONE TRUE FRAMEWORK?

DEVELOPMENT IS HARD

RAILS



django

ONE TRUE FRAMEWORK?

DEVELOPMENT IS HARD

Embark

TRUFFLE

ONE TRUE FRAMEWORK?

DEVELOPMENT IS HARD

We run *Ganache* alongside *Truffle* a new console window. This starts a new, local blockchain instance powered by Ganache. Why use it?

- Simulates full Ethereum client behavior
  - Accounts (addresses, private keys, etc.)
- Crucial debugging and logging information
- Block explorer and mining controls (block times, etc.)
- Makes developing Ethereum applications much faster

INSTALLATION

A NECESSARY STEP

```
// Make sure to have npm v5.3.0 and node v8.3.0 installed
$ npm install -g truffle

// Command line interface, github.com/trufflesuite/ganache-cli/blob/master/README.md
$ npm install -g ganache-cli

// Ganache GUI installation: truffleframework.com/ganache/
```

DIRECTORY LAYOUT

A NECESSARY STEP

- `Contracts:` contains our source code
- `Migrations:` add scripts for staging deployment tasks
- `Test:` where you test your contracts for vulnerabilities

2 MORE SOLIDITY

BLOCKCHAIN AT BERKELEY

GLOBAL VARIABLES

this

```
this; // address of contract

this.balance; // often used at end of contract life to send remaining balance to party
this.someFunction(); // calls func externally via call, not via internal jump
```

GLOBAL VARIABLES

msg, tx

```
// msg - Current message received by the contract
msg.sender; // address of sender
msg.value; // amount of ether provided to this contract in wei
msg.data; // bytes, complete call data
msg.gas; // remaining gas

// tx - This transaction
tx.origin; // address of sender of the transaction, will always be a user
tx.gasprice; // gas price of the transaction
```

GLOBAL VARIABLES

block, storage

```
// block - Information about current block
now; // current time (approximately), alias for block.timestamp (uses Unix time)
block.number; // current block number
block.difficulty; // current block difficulty
block.blockhash(1); // hash of the given block - returns bytes32, only works for most recent
256 blocks
block.gasLimit();

// storage - Persistent storage hash
storage['abc'] = 'def'; // maps 256 bit words to 256 bit words
```

FUNCTIONS

```
// Functions can return many arguments, and by specifying returned
// arguments name don't need to explicitly return
function increment(uint x, uint y) returns (uint x, uint y) {
   x += 1;
   y += 1;
}
// Call previous function
uint (a,b) = increment(1,1);
```

FUNCTIONS

```
// 'constant' indicates that function does not/cannot change persistent
vars
// Constant functions execute locally, not on blockchain
uint y;

function increment(uint x) constant returns (uint x) {
    x += 1;
    y += 1; // this line would fail
    // y is a state variable, and can't be changed in a constant function
}
```

FUNCTIONS

VISIBILITY

```
// These can be placed where 'constant' is, including:
// public - visible externally and internally (default)
// external - only visible externally
// private - only visible in the current contract
// internal - only visible in current contract, and those deriving from it

// Functions hoisted - and can assign a function to a variable
function a() {
    var z = b;
    b();
}

function b() {

}

// Prefer loops to recursion (max call stack depth is 1024)
```

FUNCTIONS
FALLBACK

```
// Fallback function - Called if other functions don't match call or
// sent ether without data
// Typically, called when invalid data is sent
// Added so ether sent to this contract is reverted if the contract fails
// otherwise, the sender's money is transferred to contract
function () {
    revert(); // reverts state to before call
}
```

EVENTS

EVENTS ARE AMAZING

```
// Declare
event LogSent(address indexed from, address indexed to, uint amount);

// note capital first letter

// Call
Sent(from, to, amount);
// For an external party (a contract or external entity), to watch:
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
}
// Common paradigm for one contract to depend on another (e.g., a
// contract that depends on current exchange rate provided by another)
```

MODIFIERS

MODIFIERS ARE ALSO AMAZING

```
// C. Modifiers
// Modifiers validate inputs to functions such as minimal balance or
user auth;
// similar to guard clause in other languages

// '_' (underscore) often included as last line in body, and indicates
// function being called should be placed there
modifier onlyAfter(uint _time) { if (now <= _time) throw; _ }


modifier onlyOwner { if (msg.sender == owner) _ }
// commonly used with state machines


modifier onlyIfState (State currState) { if (currState != State.A) _ }
```

MODIFIERS

MODIFIERS ARE ALSO AMAZING

```
// Append right after function declaration
function changeOwner(newOwner)
onlyAfter(someTime)
onlyOwner()
onlyIfState(State.A)
{
    owner = newOwner;
}

// underscore can be included before end of body,
// but explicitly returning will skip, so use carefully
modifier checkValue(uint amount) {

    _
    if (msg.value > amount) {
        uint amountToRefund = amount - msg.value;
        if (!msg.sender.send(amountToRefund)) {
            throw;
        }
    }
}
```

LOOPS

LOOPS ARE DANGEROUS

```
// All basic logic blocks work - including if/else, for, while, break, continue
// return - but no switch

// Syntax same as javascript, but no type conversion from non-boolean
// to boolean (comparison operators must be used to get the boolean val)

// For loops that are determined by user behavior, be careful - as contracts
have a maximal amount of gas for a block of code - and will fail if that is
exceeded
// For example:
for(uint x = 0; x < refundAddressList.length; x++) {
    if (!refundAddressList[x].send(SOME_AMOUNT)) {
       throw;
    }
}
```

EXTERNAL CONTRACTS

USAGE

```
contract InfoFeed {function info() returns (uint ret) { return 42; }}

contract Consumer {
    InfoFeed feed; // points to contract on blockchain


    function setFeed(address addr) { // Set feed to existing contract instance
        feed = InfoFeed(addr);// automatically cast, be careful; constructor is not called
    }

    function createNewFeed() { // Set feed to new instance of contract
        feed = new InfoFeed(); // new instance created; constructor called
    }

    function callFeed() {
        // final parentheses call contract, can optionally add
        // custom ether value or gas
        feed.info.value(10).gas(800)();
    }
}
```

INHERITANCE

USAGE

```
// Order matters, last inherited contract (i.e., 'def') can override parts of
// previously inherited contracts
contract MyContract is abc, def("a custom argument to def") {

// Override function
    function z() {
        if (msg.sender == owner) {
            def.z(); // call overridden function from def
            super.z(); // call immediate parent overriden function
        }
    }
}

// abstract function
function someAbstractFunction(uint x);
// cannot be compiled, so used in base/abstract contracts
// that are then implemented
```

IMPORTS

USAGE

```
// C. Import

import "filename";
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol";
import "github.com/Arachnid/solidity-stringutils/strings.sol";
// Importing under active development
// Cannot currently be done at command line
```

REVERT, REQUIRE, ASSERT

USAGE

The revert function can be used to flag an error and revert the current call

The require function should be used to ensure valid conditions, such as inputs, or contract state variables are met, or to validate return values from calls to external contracts.

assert acts like it might in any other programming language. Use of revert require should guarantee that your assert does not fail

REVERT, REQUIRE, ASSERT

USAGE

```
pragma solidity ^0.4.0;

contract Sharer {
    function sendHalf(address addr) payable returns (uint balance) {
        require(msg.value % 2 == 0); // Only allow even numbers
        uint balanceBeforeTransfer = this.balance;
        addr.transfer(msg.value / 2);
        // Since transfer throws an exception on failure and
        // cannot call back here, there should be no way for us to
        // still have half of the money.
        assert(this.balance == balanceBeforeTransfer - msg.value / 2);
        return this.balance;
    }
}
```

REVERT, REQUIRE, ASSERT

USAGE

assert(bool condition):

throws if the condition is not met - to be used for internal errors. (Generally used to stop very bad things from happening.)

require(bool condition):

throws if the condition is not met - to be used for errors in inputs or external components.

revert():

abort execution and revert state changes

SELFDESTRUCT

USAGE

```
// selfdestruct current contract, sending funds to address (often creator)
selfdestruct(SOME_ADDRESS);

// removes storage/code from current/future blocks
// helps thin clients, but previous data persists in blockchain

// Common pattern, lets owner end the contract and receive remaining funds
function remove() {
    if(msg.sender == creator) { // Only let the contract creator do this
        selfdestruct(creator); // Makes contract inactive, returns funds
    }
}

// May want to deactivate contract manually, rather than selfdestruct
// (ether sent to selfdestructed contract is lost)
```

STORAGE

TIPS

- Storage is expensive because it is permanent
- Things like multidimensional arrays are expensive
- Store things OFF THE BLOCKCHAIN **unless absolutely necessary**

STORAGE
TIPS

THERE ARE NO SECRETS ON THE BLOCKCHAIN

All data on the blockchain can be easily viewed by anyone in the world

UNITS OF TIME

USAGE

```
// Currency is defined using wei, smallest unit of Ether
uint minAmount = 1 wei;
uint a = 1 finney; // 1 ether == 1000 finney
// Time units
1 == 1 second
1 minutes == 60 seconds

// Can multiply a variable times unit, as units are not stored in a variable
uint x = 5;
(x * 1 days); // 5 days

// Careful about leap seconds/years with equality statements for time
// (instead, prefer greater than/less than)
```

STYLE
GUIDELINES

Quick summary:

- 4 spaces for indentation
- Two lines separate contract declarations (and other top level declarations)
- Avoid extraneous spaces in parentheses
- Can omit curly braces for one line statement (if, for, etc)
- else should be placed on own line

SEE YOU NEXT TIME

Smart Contract Security

Integer Over/Underflow

Short Address Attacks

Visibility Modifiers

Fallback Function

`delegatecall` and External Calls

Security Tools

Past Hacks