BUILDING DECENTRALISED APPLICATIONS WITH

ethereum

CHRISTOPHER GILBERT

Building Decentralized Applications with Ethereum

Christopher Gilbert

This book is for sale at http://leanpub.com/decentralizedapplicationswithethereum

This version was published on 2016-08-20

This book is dedicated to my parents, who taught me the most important lessons in life with the patience that only a teacher can have; How to live my life, how to laugh at myself, how to respect others, and above all else to love learning and teaching others.

Contents

Preface

I started my career in the games industry, thinking I would become a graphics specialist writing special effects and GPU shaders. I wasn't even 10 years old when I decided I wanted to spend my working life making video games, partly out of selfish self-interest. At that time the various platform vendors had locked down their hardware systems, making them 'tamper-proof' (read: copy protected). Casting caution aside, I bought a few parts for Â£30 or so, opened up my console (breaking the law!), and with very poor, unpracticed soldering skills I connected the few solder points and voila! I (much later) realised that I had just turned a Â£99 console into one the professionals were being charged Â£10,000 for! While I was at University I ported a certain popular open-source graphics rendering engine to the console and wrote a few good looking demos, and that work landed me my first job. Everything was going smoothly until I committed a fatal mistake and unwittingly admitted that I know what a TCP socket is. With a wave of a pre-written contract the blessing was bestowed upon me: network specialist. So with all the optimism of youth, and a little mentoring from some of the greatest unsung heroes I have ever had the privilege of working with, I set about writing my first P2P networking library. Little did I realize then, but that sequence of events would set me on a journey that would take me to perhaps the greatest digital frontiers in human history.

Throughout all the years I've been working as a software engineer, there has never been a technology breakthrough that I thought had more potential to change people's lives for the better than Bitcoin. In 2008 Satoshi Nakamoto introduced to the world his creation, which borrowed a few ideas from previous attempts at a cryptocurrency (of which Hashcash deserves an honorable mention), but the real innovation was the introduction of a deceptively simple data structure called a merkle tree. A merkle tree is a primitive data structure where each node is referenced by a cryptographic hash. Bitcoin gained the media spotlight thanks to the association between it and the Silk Road, an illegal marketplace in the dark web where vendors and customers preferred Bitcoin due to the anonymity the currency afforded, however it is important not to marginalise Bitcoin for this reason, as the impact of a trust-less currency is far more profound. There have been many cryptocurrencies created since Bitcoin was first introduced, each of them having differing implementations, some of the more popular ones are Dogecoin, Litecoin, and Zcash. The Bitcoin ecosystem has even given way to start-ups focused on any imaginable application of the cryptocurrency, including exchanges, wallets, point of sale payments, and a lot more besides!

Ethereum is the latest and greatest thinking from a long line of cryptocurrencies, created by the brilliant Vitalik Buterin. The key difference between Ethereum and Bitcoin is the introduction of a turing complete programming language designed for expressing so-called 'smart-contracts'. These contracts code that run on the blockchain and define the conditions under which transactions are .. well .. transacted. They benefit from being verified by Ethereum miners and cryptographically sealed on the blockchain, providing verifiable proof that a transaction took place. The language of Smart-Contracts is called Solidity, a blockchain aware and so-called Contract Oriented Programming Language (Contract Oriented, get it?). Whereas Bitcoin only supported a hard-coded set of

transactions, in Solidity we can create contracts that express any conceivable type of transaction, from simple peer to peer transfers, to loans, and even distributed autonomous organisations (DAO). Despite these amazing advancements, Ethereum is at its heart based on the same fundamental technology as Bitcoin, and even shares many of the same principles, but with Ethereum we can create almost any conceivable type of transaction, opening new opportunities to automate industry verticals that previously were impossible.

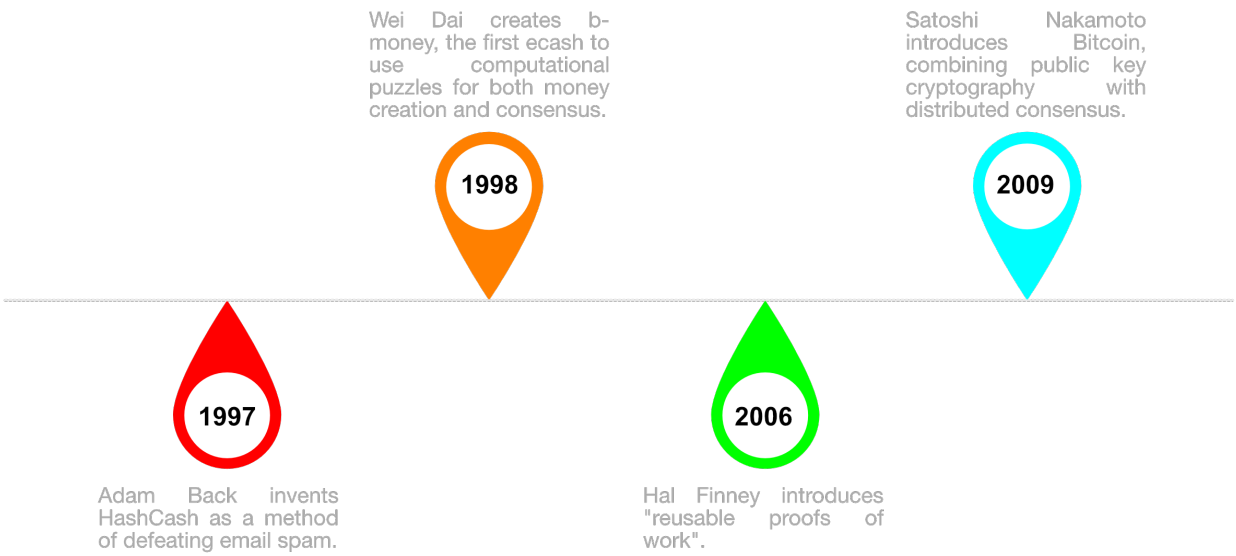
This coming together of old and new technology is ushering in a new trust-less era, which in many ways could be thought of as being the first generation of this new technology, but many are called it Web 3.0. Much like how when the Oxford printing press was created, and used to make the world's first print edition of the Christian Bible, it was only after it became the best-selling book in the world that everyone realised that this technology could be used to automate the production of any kind of book. Or when email was first introduced to the world, everyone started using it because it meant that they could send messages to each other instantly and for free, never mind that to get email you needed to invent the Internet first. So the first generation of applications were little more than the digitization of things people had in real life, and it wasn't until the technology progressed that we realised that actually we could create a completely new type of product that previously no-one could even think of. We're still in the early stages of understanding what we can create with Ethereum, and that, my dear reader, is where you come in.

It is my sincere hope that this book will inspire a new generation of bedroom hackers, willing to bend (or break!) a few rules (for the right reasons!), to change people's lives for the better, challenge the status quo, create brilliant and life changing software, and above all else have fun doing it! It is my honor to teach you the first steps on your journey.

Introduction to Ethereum Concepts

Ethereum is to date the most advanced blockchain platform ever created. Founded by Vitalik Buterin who wrote the Ethereum White Paper and leads the Ethereum Foundation which funds Ethereum development. The foundation employs developers from around the world, but equally importantly, Ethereum benefits from several VC-funded and financial technology teams using Ethereum for commercial development. With public chat channels hosting hundreds of developers, regular meetups around the world, and even an annual Ethereum conference, it is safe to say that there is a lot of momentum behind Ethereum. But how did we get to this point, and what is it that differentiates Ethereum from the competition? In this chapter we're going to delve deep into the history, technology and design decisions that underpin this disruptive technology.

A Brief History of Cryptocurrencies



History of cryptocurrencies

It may surprise some readers to learn that the concept of electronic cash is not a new idea. The first attempts at electronic cash were conducted back in the 1980's, and used Chaumian Blinding to provide users with a high degree of privacy. Unfortunately, these systems ultimately failed to gain widespread use due to their dependency on centralized servers.

In 1997, Adam Back invented the concept of using computational puzzles to defeat email spam in a system called HashCash. Over two decades later the same algorithm would be repurposed by Bitcoin

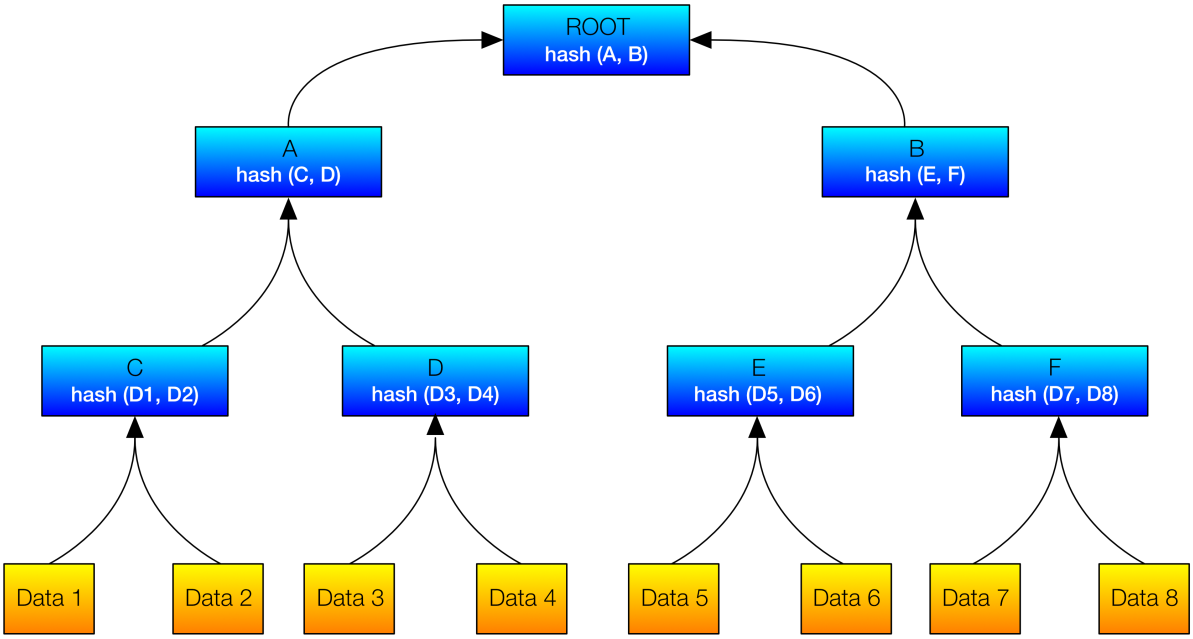
for it's Proof Of Work algorithm, intended to prevent malicious users from attacking the network in something called a Sybil attack.

Just one year later in 1998, Wei Dai invented a new type of electronic cash called B-Money, and was the first to introduce the idea of using computational puzzles to both create money as well as a decentralized consensus method. Unfortunately the proposal did not present sufficient details of how the consensus could actually be implemented.

Hal Finney was the first person to introduce the concept of so-called "reusable proofs of work" in 2006. Building on the concepts introduced by B-Money, the idea was to combine Adam Back's HashCash puzzles with money creation, but again this idea was not successful due to the reliance on centralized servers.

In 2009 Satoshi Nakamoto was the first to implement a decentralized electronic currency, using public key cryptography to establish ownership, and a consensus algorithm for keeping track of who owns what, called "proof of work". The latter was a breakthrough because it solved two problems. First it provided a method for nodes to agree on the order of updates to the state of the distributed ledger, and second it solved the problem of Sybil attacks. It did this by introducing an economic incentive to participate with the network, since the influence of a node in the consensus process is proportional to the computing power of the node.

Merkle Trees



Merkle Tree

An important scalability feature of Bitcoin is that the block is stored in a multi-level data structure. The "hash" of a block is actually only the hash of the block header, a roughly 200-byte piece of data that contains the timestamp, nonce, previous block hash and the root hash of a data structure called the Merkle tree storing all transactions in the block. A Merkle tree is a type of binary tree, composed of a set of nodes with a large number of leaf nodes at the bottom of the tree containing the underlying data, a set of intermediate nodes where each node is the hash of its two children, and finally a single root node, also formed from the hash of its two children, representing the "top" of the tree. The purpose of the Merkle tree is to allow the data in a block to be delivered piecemeal: a node can download only the header of a block from one source, the small part of the tree relevant to them from another source, and still be assured that all of the data is correct. The reason why this works is that hashes propagate upward: if a malicious user attempts to swap in a fake transaction into the bottom of a Merkle tree, this change will cause a change in the node above, and then a change in the node above that, finally changing the root of the tree and therefore the hash of the block, causing the protocol to register it as a completely different block (almost certainly with an invalid proof of work).

The Merkle tree protocol is arguably essential to long-term sustainability. A "full node" in the Bitcoin network, one that stores and processes the entirety of every block, takes up about 15 GB of disk space in the Bitcoin network as of April 2014, and is growing by over a gigabyte per month. Currently, this is viable for some desktop computers and not phones, and later on in the future only businesses and hobbyists will be able to participate. A protocol known as "simplified payment verification" (SPV) allows for another class of nodes to exist, called "light nodes", which download the block headers, verify the proof of work on the block headers, and then download only the "branches" associated with transactions that are relevant to them. This allows light nodes to determine with a strong guarantee of security what the status of any Bitcoin transaction, and their current balance, is while downloading only a very small portion of the entire blockchain.

Ethereum Accounts

In Ethereum there are two types of accounts: - `external`: Externally owned and controlled by private keys, these accounts have no code, and the owner can create and send transactions. - `contract`: Contract accounts are controlled by code which is triggered by messages. Contracts can also send messages, and create new contracts.

images/chapter1/1-3.png

Ethereum accounts contain the following four fields: - `nonce`: A counter which ensures a transaction may only be processed exactly once. - `balance`: The current ETH balance of the account. - `code`: The account's contract code (if there is any). - `storage`: The account's storage (if there is any).

Cryptocurrency Transactions

The term "transaction" is used in Ethereum to refer to the signed data package that stores a message to be sent from an externally owned account.

images/chapter1/1-4.png

Transactions contain: - `recipient`: The account the transaction is intended for. - `sender`: A signature representing the originator of the transaction. - `amount`: The amount of ETH to transfer from the sender to the recipient. - `data`: An optional field to pass data to the smart contract. - `startgas`: A value representing the maximum number of computational steps the transaction execution is allowed to take. - `gasprice`: A value representing the fee the sender pays per computational step.

The first three are standard fields expected in any cryptocurrency. The data field has no function by default, but the virtual machine has an opcode with which a contract can access the data; as an example use case, if a contract is functioning as an on-blockchain domain registration service, then it may wish to interpret the data being passed to it as containing two "fields", the first field being a domain to register and the second field being the IP address to register it to. The contract would read these values from the message data and appropriately place them in storage.

The STARTGAS and GASPRICE fields are crucial for Ethereum's anti-denial of service model. In order to prevent accidental or hostile infinite loops or other computational wastage in code, each transaction is required to set a limit to how many computational steps of code execution it can use. The fundamental unit of computation is "gas"; usually, a computational step costs 1 gas, but some operations cost higher amounts of gas because they are more computationally expensive, or increase the amount of data that must be stored as part of the state. There is also a fee of 5 gas for every byte in the transaction data. The intent of the fee system is to require an attacker to pay proportionately for every resource that they consume, including computation, bandwidth and storage; hence, any transaction that leads to the network consuming a greater amount of any of these resources must have a gas fee roughly proportional to the increment.

Smart-Contract Messages

Contracts have the ability to send "messages" to other contracts. Messages are virtual objects that are never serialized and exist only in the Ethereum execution environment.

images/chapter1/1-5.png

A message contains: - `sender`: The sender of the message. - `recipient`: The recipient of the message. - `amount`: The amount of ether to transfer. - `data`: An optional data field. - `startgas`: A STARTGAS value.

Essentially, a message is like a transaction, except it is produced by a contract and not an external actor. A message is produced when a contract currently executing code executes the `CALL` opcode, which produces and executes a message. Like a transaction, a message leads to the recipient account running its code. Thus, contracts can have relationships with other contracts in exactly the same way that external actors can.

Note that the gas allowance assigned by a transaction or contract applies to the total gas consumed by that transaction and all sub-executions. For example, if an external actor A sends a transaction to B with 1000 gas, and B consumes 600 gas before sending a message to C, and the internal execution of C consumes 300 gas before returning, then B can spend another 100 gas before running out of gas.

Code Execution

The code in Ethereum contracts is written in a low-level, stack-based bytecode language, referred to as "Ethereum virtual machine code" or "EVM code". The code consists of a series of bytes, where each byte represents an operation. In general, code execution is an infinite loop that consists of repeatedly carrying out the operation at the current program counter (which begins at zero) and then incrementing the program counter by one, until the end of the code is reached or an error or STOP or RETURN instruction is detected. The operations have access to three types of space in which to store data:

- `stack`: a last-in-first-out container to which values can be pushed and popped

- `memory`: an infinitely expandable byte array
- `storage`: a key/value store. Unlike stack and memory, which reset after computation ends, storage persists for the long term.

The code can also access the value, sender and data of the incoming message, as well as block header data, and the code can also return a byte array of data as an output.

The formal execution model of EVM code is surprisingly simple. While the Ethereum virtual machine is running, its full computational state can be defined by the tuple (block_state, transaction, message, code, memory, stack, pc, gas), where block_state is the global state containing all accounts and includes balances and storage. At the start of every round of execution, the current instruction is found by taking the pcth byte of code (or 0 if pc >= len(code)), and each instruction has its own definition in terms of how it affects the tuple. For example, ADD pops two items off the stack and pushes their sum, reduces gas by 1 and increments pc by 1, and SSTORE pushes the top two items off the stack and inserts the second item into the contract's storage at the index specified by the first item. Although there are many ways to optimize Ethereum virtual machine execution via just-in-time compilation, a basic implementation of Ethereum can be done in a few hundred lines of code.

Sybil Attack

With Bitcoin, however, nodes are numerous, mostly anonymous, and can enter or leave the system at any time. Unless one puts in careful thought, such a system would quickly run into what is known as a Sybil attack, where a hostile attacks simply creates five times as many nodes as the rest of the network combined, whether by running them all on the same machine or rented virtual private server or on a botnet, and uses this supermajority to subvert the network. In order to prevent this kind of attack, the only known solution is to use a resource-based counting mechanism. For this purpose, Bitcoin uses a scheme known as proof-of-work, which consists of solving problems that are difficult to solve, but easy to verify. The weight of a node in the consensus is based on the number of problem solutions that the node presents, and the Bitcoin system rewards nodes that present such solutions ("miners") with new bitcoins and transaction fees.

Turing Complete Smart Contracts

The Halting Problem

Whereas all transactions in Bitcoin are roughly the same, and thus their cost to the network can be modeled to a single unit, transactions in Ethereum are more complex, and so a transaction fee system needs to take into account many ingredients, including cost of bandwidth, cost of storage and cost of computation. Of particular importance is the fact that the Ethereum programming language is Turing-complete, and so transactions may use bandwidth, storage and computation in arbitrary

quantities, and the latter may end up being used in quantities that due to the halting problem cannot even be reliably predicted ahead of time. Preventing denial-of-service attacks via infinite loops is a key objective.

Determinism

The Ethereum Virtual Machine purposefully removes any possible sources of non-determinism. Non-determinism in this context is considered to be any programming construct that, when evaluated by a given target machine, may or may not produce the same results. Examples include: - The concept of time. Time is very rarely the same across a cluster of machines, even after synchronising the clocks. - Network access. Firewall rules mean different machines can get different responses, not to mention the massive and obvious DDOS that such a feature would produce. - Floating point numbers. This one is perhaps the hardest to believe, but

The Ethereum Virtual Machine (EVM)

The Ethereum Virtual Machine is the method by which smart contracts are actually evaluated by an Ethereum client. The formal specification of the EVM is given in the so-called Yellow Paper, authored by Dr. Gavin Wood. At a high level you can think of the EVM as being a platform independent, deterministic state machine, specialised for cryptographic calculations. The determinism is an important aspect of miners reaching consensus, otherwise if miners are not able to agree on the result of running a smart-contract, then every transaction would fail.

The EVM is designed to be as simple as possible, so there are very few op-codes, but usually developers do not need to think about the EVM, as high-level languages such as Solidity generate low-level EVM code. The resulting generated code is generally very compact and space efficient, which is essential given that smart-contracts are propagated across the entire network when they are deployed and run by the clients.

Of direct relevance to developers is the EVM memory model, which includes 3 types of storage: stack (a LIFO stack of 32-byte values), memory (infinitely expandable temporary storage), and storage (permanent storage that persists between EVM runs). For the readers who are familiar with the machine architecture of current computers (eg. Intel x64, ARM, PPC), it may come as a shock to find out that the word size of the EVM is 32-bytes, chosen because 32-byte words are common in many cryptographic algorithms.

Finally the EVM has no stack size limit, justified by the fact that block-level gas limits will restrict the consumption of stack space, and just as importantly there is a 1024 call-depth limit.

Conclusion

In this chapter we took a look at the underlying technology and design decisions that differentiate Ethereum from other cryptocurrencies.

Writing Your First Smart Contract

In this chapter we are going to delve right into Ethereum with an opinionated approach that will get you started with the basic tools you need to create your first Distributed Application (DApp).

Installation And Setup

The Ethereum Foundation publishes the specifications for Ethereum clients on their Github page, and funds development of official clients written in Go (go-ethereum), C++ (cpp-ethereum) and Python (py-ethereum). There are also community contributed clients in many other languages, including Ruby, Java, Haskell and Rust. However, before we try out a client, we are going to take a look at Truffle by Consensys, which is a development environment, testing framework and asset pipeline for Ethereum. Truffle will make it easier for us to get started, however you will need to make sure you have Node.js installed before you proceed. Once you have Node installed and the accompanying `npm` package manager, you simply need to run `npm install -g truffle` to install the framework and its dependencies.

First we will use truffle to create a new project. Start by making a new folder on your system via a terminal, change into the new folder and run `truffle init`: `$ mkdir chapter_1 && cd chapter_1 $ truffle init`

By default, truffle will create a project structure with the following folders: - `app/` should contain application files, JavaScript, CSS and HTML should all go in here. - `contracts/` should contain smart contracts written in `Solidity`, which we will go into more detail soon. - `migrations/` should contain scriptable deployment files. - `test/` should contain unit tests written using Mocha and Chai, which we will use soon. - `truffle.js` is the main configuration file used by Truffle.

Compiling Smart-Contracts

Now that we have created our new project, let's go ahead and compile the smart contract provided by Truffle with the command `truffle compile`. This command will create a new directory `build/contracts` which contains the output of the compile command, and should now contain some JavaScript files. This generated code will allow us to do several useful things with our smart contracts, which we will go into shortly, but before we do that let's put the `MetaCoin.sol` contract under the microscope.

As this is the first Solidity code we have seen, I will break the contract into pieces to explain them in more detail. Readers who are already familiar with JavaScript will see many similarities with Solidity, but I will try to explain the functional aspects of the contract for everyone, and save a deep-dive analysis of Solidity for later chapters.

```
import "ConvertLib.sol";
```

This first line instructs Solidity to import the library functions in ConvertLib.sol. I don't include the contents here for clarity, but ConvertLib contains a function `convert` which takes an amount and a conversion rate as parameters, and returns the converted amount.

```
contract MetaCoin {
```

Smart Contracts always start with the keyword `contract` and an identifier, in this case the identifier is `MetaCoin`, which we can use to refer to the contract. The opening curly brace denotes that everything between it and a matching closing curly brace is the contract code. We use indentation to help us visually track opening and closing braces.

```
mapping (address => uint) balances;
```

This line specifies a member variable called `balances`. Solidity is a strongly-typed language which means that variables must have a type given to them. Here the type is `mapping(address => uint)`. A mapping is a relational type, from keys to values, which in this instance the key is an `address` which is a cryptographic hash (SHA3), and the value is a `uint`. Addresses refer to user accounts, and a uint is a type of unsigned integer, which means the value must always be positive. We will see how this mapping is used later.

```
event Transfer(address indexed _from, address indexed _to, uint256 _value);
```

Next we specify an `event`, which is how in Solidity we use the logging facility of the Ethereum Virtual Machine. When called, events cause the arguments to be stored in the transaction's log. In this instance we are storing three parameters, `_from` which is of type `address`, `_to_` which is another `address`, and `_value` which is a `uint256` (256-bit unsigned integer). Notice that `_from` and `_to` have an additional attribute `indexed`. This attribute causes those arguments to be treated as log topics, rather than data.

```
function MetaCoin() {
        balances[tx.origin] = 10000;
}
```

Next we specify a special function which is a `constructor`, denoted by the function having the same name as the contract itself. Generally constructors are used to initialize contract member variables to their initial state once the contract has been deployed. Here we use a special variable called `tx.origin` which is of type `address` and contains the sender of the transaction, and it is used to insert a key / value pair into the balances member, where the key is the sender, and the value is 10000.

```
function sendCoin(address receiver, uint amount) returns(bool sufficient) {
        if (balances[msg.sender] < amount) return false;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        Transfer(msg.sender, receiver, amount);
        return true;
}
```

There's a few things going on here, but let's start with the function signature. This function takes two parameters, `receiver` and `amount`, and returns a type of `bool` (true or false) called `sufficient`. The first line of the function uses a new special variable `msg.sender`, which is, intuitively, the address of the message sender. If the message sender does not have sufficient funds, then the function returns false immediately. Otherwise the balances of the message sender and the receiver are updated, a Transfer event is appended to the transaction log, and then the function returns true.

```
function getBalanceInEth(address addr) returns(uint){
        return ConvertLib.convert(getBalance(addr),2);
}
```

This function demonstrates the use of the imported library function `ConvertLib.convert`. Here the parameters are the result of calling `getBalanced` (defined beneath this function), and the uint `2`. Note how the imported library function can be called by simply referring to the name of the function and supplying the necessary parameters that the function requires.

```
function getBalance(address addr) returns(uint) {
        return balances[addr];
}
```

Finally the `getBalance` function returns the balance of the account given by the parameter `addr`. Here we lookup the address in the balances mapping, and return the value contained.

Testing Smart-Contracts

Truffle provides testing facilities via the JavaScript testing library Mocha and assertions library Chai. The biggest difference between Mocha and Truffle tests is that Truffle provides the special function `contract()` which works like Mocha's `describe()` function, with the following modifications: - Before each test function, the contract a redeployed to the Ethereum client to reset the contract state. - A list of available accounts is provided as a second parameter, which you can write tests against.

Note that Truffle also imports the pre-compiled contract code built earlier with `truffle compile`, which saves us having to do it manually. So let's put one of the tests generated by Truffle under the microscope.

```
contract('MetaCoin', function(accounts) {
```

First we begin by defining the test suite with the given name `MetaCoin`. This line instructs Truffle to deploy the contracts first and then run the tests specified with `it()`.

```
it("should put 10000 MetaCoin in the first account", function() {
```

Next we specify a test to check that when a contract is deployed it puts 10000 MetaCoin in the first account.

```
var meta = MetaCoin.deployed();
```

This is the first line written that directly references the generated contract code. `MetaCoin` is a contract proxy generated by Truffle that makes interacting with contracts simpler and more convenient. The function `MetaCoin.deployed()` will find the deployed contract and return a JavaScript object that allows us to interact with it.

```
return meta.getBalance.call(accounts[0]).then(function(balance) {
  assert.equal(balance.valueOf(), 10000, "10000 wasn't in the first account");
});
```

There's a few things going on here, so let's break it down. First of all we call `meta.getBalance.call()` and pass in the value of `accounts[0]` which returns a `Promise`, note that `call()`'s are read-only, as opposed to transactions, which change the state of the network. We'll dip into these topics in more detail in the next chapter. When the promise yields a result, we then use the Chai function `assert.equal()`, and the first parameter is the value returned by the promise `balance`, the second parameter is the expected value, and the third parameter is a string description of what was being asserted.

To run the tests, we first need to install `EthereumJS TestRPC`. The reason we use this particular tool is because it is significantly faster than testing against the go-ethereum or cpp-ethereum clients, and is ideal for use during development, though you should test against one of the official clients before deploying to a public network. To install TestRPC, simply run `npm install -g ethereumjs-testrpc`.

After you have installed TestRPC, run it in another terminal with `testrpc` and then with TestRPC still running, run `truffle test`, which will run the Mocha tests against the client with the settings specified in `truffle.js`, which defaults to localhost:8545. You should see the output from Mocha printed to your terminal:

```
Contract: MetaCoin
  ✓ should put 10000 MetaCoin in the first account (39ms)
  ✓ should call a function that depends on a linked library   (57ms)
  ✓ should send coin correctly (119ms)


3 passing (2s)
```

Migrations

Migrations are how smart contracts are deployed to the Ethereum network. The assumption is that deployment needs change over time, and as our contracts evolve we will need to create new scripts to deploy updates to the blockchain. A history of all migrations is preserved on the blockchain via a special Migrations contract. Let's take a look at that contract now, in `contracts/Migrations.sol`.

```
contract Migrations {
```

First we start with the familiar contract definition, in this case defining a contract called `Migrations`.

```
address public owner;
```

Here we see a familiar storage variable declaration `owner` of type `address`, but with a special visibility attribute `public`. This creates an automatic accessor function in the external interface of the contract, meaning external functions can call `Migrations.owner()` to retrieve the value of owner in storage. In this contract, owner will be used to store the address of the account that created the contract.

```
uint public last_completed_migration;
```

Next we have a fairly uninteresting storage variable declaration called `last_completed_migration`. This variable is intended to store the identifier of the last completed migration.

```
modifier restricted() {
  if (msg.sender == owner) _
}
```

Now we get to something new, this is an example of a Function Modifier. Function modifiers change the behaviour of functions, in this instance the function modifier `restricted` contains a check to see if the value of the special variable `msg.sender` is the same as the value of the storage variable `owner`. The underscore _ denotes the inclusion of the remainder of the function body to which the modifier is applied. The effect of this particular modifier is to only run the function it is applied to if the message sender also owns the contract.

```
function Migrations() {
  owner = msg.sender;
}
```

Here in the contract constructor the storage variable `owner` is assigned the value of `msg.sender`. Because this variable is set in the constructor, it is only set once when the contract is created.

```
function setCompleted(uint completed) restricted {
  last_completed_migration = completed;
}
```

Here the function `setCompleted` has the restricted modifier, which as we have already discussed will only allow modified code to run if the function is called by the owner. In this case, if the message sender is indeed the same as the owner of the contract then the last_completed_migration variable will be updated, otherwise nothing will happen.

```
function upgrade(address new_address) restricted {
  Migrations upgraded = Migrations(new_address);
  upgraded.setCompleted(last_completed_migration);
}
```

Here we have a function which provides the contract upgrade mechanism. This function takes the address of the new contract, and creates a handle to the contract from the new address and calls the setCompleted function on the new contract.

This contract is deployed via the following migration script in `migrations/1_initial_migration.js`.

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

After this script has been run, subsequent migrations can be created with increasing numbered prefixes. Truffle uses the prefix and the Migrations contract to deduce which migration scripts to run. Let's analyse the migration script to deploy the MetaCoin contract in `2_deploy_contracts.js`.

```
module.exports = function(deployer) {
```

First we have the usual boilerplate to export the migration function, which takes a deployer object as an argument, and is provided by Truffle when the script is run.

```
deployer.deploy(ConvertLib);
```

First we deploy the ConvertLib library code. This is necessary because the MetaCoin contract depends on ConvertLib, and so it must be deployed first.

```
deployer.autolink();
```

Next we tell the deployer to automatically link any required libraries to the contracts being deployed, if the addresses of those libraries are available.

```
deployer.deploy(MetaCoin);
```

Finally we deploy the MetaCoin contract. As we have enabled autolinking, the address of the ConvertLib library is automatically linked to the MetaCoin contract. Alternatively we could have used `deployer.link(library, destinations)`, but the result is the same in either case.

Migrations can be run by running `truffle migrate` on the command line, though you must keep your local Ethereum client running when you run this command, when it completes successfully for the first time you should see the following output printed to your terminal:

```
Running migration: 1_initial_migration.js
  Deploying Migrations...
  Migrations: 0xa7b7a17675e713226a98e03bec9b20b227455ff6
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying ConvertLib...
  ConvertLib: 0xb231a058f9f9d2a8955fb6c393fb66430c050a48
  Linking ConvertLib to MetaCoin
  Deploying MetaCoin...
  MetaCoin: 0xf56766a3284af35fb8939db7ffe14e4e598e7511
Saving successful migration to network...
Saving artifacts...
```

Note that if you run `truffle migrate` a second time, no migration scripts will be run. This is due to the Migrations contract, stored on the blockchain, which Truffle will check to see which migrations need to be run. The second time you run the command, the `Migrations.last_completed_migration()` will return `2`, and so Truffle will only run newer scripts as denoted by the numeric prefix `3_sample_description.js` and so on.

Web UI

The Web UI is how end-users will interact with your distributed application. For this purpose, the Ethereum Foundation develops the `web3` library, which is a client library for interacting with the blockchain via a standard JSON RPC interface exposed by the Ethereum client. The generated contract code uses web3 under the hood, but otherwise hides it behind a convenience interface. Let's take a look at `app/javascripts/app.js` and see how to use the generated contracts.

```
function refreshBalance() {
  var meta = MetaCoin.deployed();

  meta.getBalance.call(account, {from: account}).then(function(value) {
    var balance_element = document.getElementById("balance");
    balance_element.innerHTML = value.valueOf();
  }).catch(function(e) {
    console.log(e);
    setStatus("Error getting balance; see log.");
  });
};
```

As might be evident to the observative reader, the same functions used in the unit tests are available in the browser JavaScript. Here we use `MetaCoin.deployed()` to get a handle to the previously deployed contract on the blockchain, and then we can call `meta.getBalance.call()` which is a read-only request to the blockchain to retrieve the value of `getBalance()`. The retrieved value is then written into the browser DOM, or if an error occurs it is written to the console log. The overall result of this function is to retrieve the balance of the current user from the blockchain and display it.

```
function sendCoin() {
  var meta = MetaCoin.deployed();

  var amount = parseInt(document.getElementById("amount").value);
  var receiver = document.getElementById("receiver").value;

  setStatus("Initiating transaction... (please wait)");

  meta.sendCoin(receiver, amount, {from: account}).then(function() {
    setStatus("Transaction complete!");
    refreshBalance();
  }).catch(function(e) {
    console.log(e);
    setStatus("Error sending coin; see log.");
```

```
  });
};
```

The `sendCoin()` function retrieves the values provided by the user in a web form, and then calls `meta.sendCoin()` to write the values to the blockchain in a `transaction`. Note that transactions are therefore distinct from calls, in the way that calls are read-only (and therefore do not cost any gas).

```
window.onload = function() {
  web3.eth.getAccounts(function(err, accs) {
    if (err != null) {
      alert("There was an error fetching your accounts.");
      return;
    }

    if (accs.length == 0) {
      alert("Couldn't get any accounts! Make sure your Ethereum client is config\
ured correctly.");
      return;
    }

    accounts = accs;
    account = accounts[0];

    refreshBalance();
  });
}
```

Truffle comes with a build tool that makes it easy and convenient to build the web interface for your project, including ES6 and JSX, inclusion of `web3`, SASS support, and UglifyJS for minimising your JavaScript assets. For now we will simply run `truffle build`. This command creates the following: - `build/images` containing all image files used by your DApp from `app/images`. - `build/app.css` containing the pre-compiled CSS from `app/stylesheets`. - `build/app.js` containing the minified Javascript from `app/javascripts`. - `build/index.html` containing the html interface from `app/index.html`.

These paths are all defined in `truffle.js`, and can be changed to suit your application's needs and preferences. When you are ready run can run `truffle serve` to launch a web server on your localhost, and you can visit the interface in a web browser if you navigate to http://localhost:8080/.

Screenshot of the sample application

Conclusion

In this chapter we covered a lot of ground; We learned the basic structural elements of Solidity Smart-Contracts, how to write unit tests and run them using the TestRPC client, we learned how to write migrations and deploy smart contracts to the blockchain, and how to interact with them via JavaScript running in a web browser.

Truffle is a somewhat opinionated framework, but it serves as a convenient starting point. In a single chapter we now have the beginning of an Ethereum AltCoin, complete with unit tests and a web UI. We necessarily skipped over a lot of the details, all of which will be addressed in subsequent chapters. However we did cover a lot of topics, and you should now have a high-level picture of how these moving parts connect to make a distributed application. In later chapters we will build upon these concepts to build different kinds of distributed applications, all powered by Ethereum!