

Fatima Castiglione Maldonado

Introduction to Blockchain and Ethereum

Use distributed ledgers to validate digital transactions in
a decentralized and trustless manner



Packt >

Introduction to Blockchain and Ethereum

Use distributed ledgers to validate digital transactions in a decentralized and trustless manner

Fatima Castiglione Maldonado



Introduction to Blockchain and Ethereum

Copyright © 2018 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Author: Fatima Castiglione Maldonado

Technical Reviewer: Joaquim Pedro Antunes

Managing Editor: Alex Mazonowicz

Acquisitions Editor: Aditya Date

Production Editors: Samita Warang, Ratan Pote

Editorial Board: David Barnes, Ewan Buckingham, Simon Cox, Manasa Kumar, Alex Mazonowicz, Douglas Paterson, Dominic Pereira, Shiny Poojary, Saman Siddiqui, Erol Staveley, Ankita Thakur, and Mohita Vyas

First Published: September 2018

Production Reference: 1280918

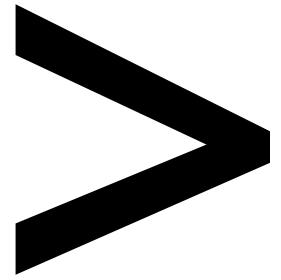
ISBN: 978-1-78961-271-4

Table of Contents

Preface	i
<hr/>	
Ethereum Blockchain	1
<hr/>	
Introduction	2
Introducing the Ethereum Blockchain	2
The Blockchain Data Structure	2
Public Key Cryptography	3
Distributed Ledgers	3
Consensus Mechanism	3
Introducing Cryptocurrencies	4
Introducing Networks and Smart Contracts	4
Cryptology and Keys	5
Traditional Codes and Cryptography	6
New Cryptography	8
Opening an Ethereum Account	10
Account Numbers and their Associated Private Keys	10
Wallets	11
Exercise 1: Creating a Wallet and Safeguarding its Information	12
Private Keys and Public Keys	13
Using your Wallet	14
Exercise 2: Getting the Toy Ether from the Rinkeby Test Network	14
The Ethereum Network, Nodes, and Mining	16
The Ethereum Network	16
Nodes	18
Mining	19

Transactions and Blocks	20
Transactions and Calls	20
Calls	20
Exercise 3: Calling the Ethereum Network	20
Transactions, Transaction Hashes, and Gas	20
Blocks and Block Hashes	24
Confirmations	26
Sending and Checking Transactions	26
Sending Transactions	27
Exercise 4: Sending and Receiving Transactions	27
Receiving Transactions	28
Checking Transactions	28
Summary	28
Learning Solidity	31
<hr/>	
Introduction	32
The Solidity Language	32
Your First Smart Contract	34
Activity 1: Creating an Ethereum Token	37
Exercise 5: Using Remix to Compile Our Token	39
Basic Solidity	40
Solidity Data Types	41
Global and Local Variables	41
Collections	42
Mappings	43
Exercise 6: Creating Our Own Collection	44
Testing Solidity	45
Exercise 7: Deploying and Testing a Smart Contract	47

Summary	57
Solidity Contracts	59
<hr/>	
Introduction	60
Your First dApp	60
Architecture of a dApp	61
Ganache	61
Exercise 8: Using MetaMask to connect to Ganache	64
Voting Contract	68
Compiling and Deploying Contracts	69
A Simple Web Page	70
Using an Oracle	72
Interface	72
Payment	74
Calculating Payments	74
Request Types	75
Functions and Getters	76
Summary	81
Index	83



Preface

About

This section briefly introduces the author, the coverage of this course, the technical skills you'll need to get started, and the hardware and software requirements required to complete all of the included activities and exercises.

About the Book

Blockchain applications provide a single-shared ledger to eliminate trust issues involving multiple stakeholders.

With the help of Introduction to Blockchain and Ethereum, you'll learn how to create distributed blockchain applications that do not depend on a central server or datacenter. The course begins by explaining Bitcoin, Altcoins, and Ethereum, followed by taking you through distributed programming using the Solidity language on the Ethereum blockchain.

By the end of this course, you'll be able to write, compile, and deploy your own smart contracts to the Ethereum blockchain.

About the Author

Fatima Castiglione Maldonado is an entrepreneur with more than 10 years of experience in the IT industry and 5 years of experience in the cryptocurrency space. Her team at Ethernity.live does contract work for crypto, and they are now also launching their own project.

"This book was written by me, Fatima Castiglione Maldonado, and co-authored by Marco Castiglione Maldonado, who wrote made corrections and amendments to most of the material. The code included in this book is based on developments done by our team at Ethernity.live (Juan Livingston, Jaime Irazabal, and Yoscar Hernandez). Big thanks to Giannella Papini and Fiona Castiglione Maldonado for their support; also, thanks to all the team at Packt."

Objectives

- Grasp blockchain concepts such as private and public keys, addresses, wallets, and hashes
- Send and analyze transactions in the Ethereum Rinkeby test network
- Compile and deploy your own ERC20-compliant smart contracts and tokens
- Test your smart contracts using MyEtherWallet
- Create a distributed web interface for your contract
- Combine Solidity and JavaScript to create your very own decentralized application

Audience

Introduction to Blockchain and Ethereum is ideal for you if you want to get to grips with blockchain technology and develop your own distributed applications with smart contracts written in Solidity. Prior exposure to an object-oriented programming language such as JavaScript is needed, as you'll cover the basics before getting straight to work.

Approach

This course thoroughly explains the technology in an easy-to-understand language while perfectly balancing theory and exercises. Each lesson is designed to build on the learnings of the previous lesson. The course also contains multiple activities that use real-life business scenarios for you to practice and apply your new skills in a highly relevant context.

Minimum Hardware Requirements

For an optimal student experience, we recommend the following hardware configuration:

- Intel Core i3 processor or equivalent
- 2 GB RAM (1.5 GB if running on a virtual machine)
- 10 GB available hard disk space
- 5400 RPM hard disk drive
- DirectX 9-capable video card (1024 x 768 or higher resolution)
- Internet connection

Software Requirements

You'll also need the following software installed in advance:

- Operating system: Windows 8 or higher (64-bit version)
- Mist (<https://github.com/ethereum/mist/releases/>)
- Truffle (<http://truffleframework.com/>)

Installing the Code Bundle

Copy the code bundle for the class to the **C:/Code** folder.

Additional Resources

The code bundle for this course is also hosted on GitHub at: <https://github.com/TrainingByPackt/Introduction-to-Blockchain-and-Ethereum>.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You should name the file **index.js**."

A block of code is set as follows:

```
/* Contract data: array with balances and initial number of tokens */
mapping (address => uint256) public balanceOf;
uint initialSupply = 1000000 public;
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Enter the receiver address in the **To Address** field."

1

Ethereum Blockchain

Learning Objectives

By the end of this lesson, you will be able to:

- Describe key blockchain concepts such as keys, cryptology, networks, nodes, and mining
- Set up and use an Ethereum account
- Send and check transactions using the Ethereum network

This lesson will start with a focus on the theory behind cryptology and blockchain technologies, then we will perform a practical exercise, setting up a wallet, and executing transactions.

Introduction

Only a few years ago, monetary transactions needed a central control authority to be sent and received. This central control authority created and maintained a database of transactions, and could both modify and block users' transactions.

In 2009, **Bitcoin** ushered in the first truly usable distributed transaction ledger, which has started to impact traditional monetary systems.

Note

You can get a nice view of the cryptocurrency ecosystem at <https://coinmarketcap.com/all/views/all/>.

Introducing the Ethereum Blockchain

Although bitcoin has become a major buzzword in technology over the past few years, blockchain technology is more than just investment opportunities. Blockchains are peer-to-peer networks that use cryptology and distributed computers systems, and which can be used to share data and build applications. Blockchain has the potential to impact many data-focused aspects of everyday life, from banking and payments, to big data and smart contracts.

Blockchain and bitcoin are not the same thing; bitcoin is implemented using blockchain technology, but blockchain technology can be used in contexts much wider than bitcoin or cryptocurrencies. The term blockchain refers to the combination of a number of technologies, including the following:

- The blockchain data structure
- Public key cryptography
- Distributed ledgers
- Consensus mechanisms

The Blockchain Data Structure

A blockchain is a special type of database in which the data is set out and built up in successive blocks. Each of the blocks of data includes a small piece of data that verifies the content of the previous block. As a result, if an attempt is made to modify an earlier block in the chain, all of the later blocks cease to match up. The system that maintains the blockchain will be able to detect and reject the attempted modification, and this is what makes the blockchain tamper-proof.

Public Key Cryptography

The use of public key cryptography ensures that each participant in the system is uniquely identified and can validate any change to the blockchain using a cryptographically secure private key. While public key cryptography is not unique to blockchain, it is one of the essential underlying technologies, which ensures that blockchains are secure and that only authorized participants can make changes to a blockchain. It can also be used to encrypt data stored on the blockchain so that the data can only be accessed by those with the key to decrypt it.

Distributed Ledgers

Traditional ledger systems either require each participant to maintain their own decentralized ledger or they require the participants to trust a centralized ledger. The problem with decentralized ledgers is that they can be costly to maintain and to keep secure, and it may not become immediately apparent when they diverge until a transaction down the line reveals that each ledger in fact records a different version of the facts. A centralized ledger, on the other hand, requires all the parties to trust the holder of the authoritative central ledger and creates a critical vulnerability: what happens if the central ledger is hacked or a disgruntled employee deletes it? The key to a distributed ledger is that each authorized participant (a node) maintains a complete version of the ledger and each transaction. That is, each proposal to modify the ledger is sent out to all of the nodes and is only approved if a sufficient number of nodes agree that it is a valid transaction.

Consensus Mechanism

This validation of proposed changes to the blockchain is performed by the nodes in accordance with certain preset rules whereby the nodes will reach a consensus as to whether the new data entry will be permitted (for example, the nodes might conduct a check to confirm that, according to the records on the blockchain, the participant purporting to conduct a particular transaction owns the relevant asset that is the subject of that transaction). This is the consensus mechanism, and only if there is agreement between the nodes as to the validity of the transaction represented by that data entry will that data entry be permitted to be appended to the blockchain (that is, another Lego block will be added to the tower). Once that transaction has been approved, however, the updated version of the blockchain with the newly appended entry will rapidly spread throughout the system so that that all of the nodes end up with an identical version of the ledger.

This consensus mechanism means that there is a rigorous means, applied uniformly by all participants, that ensures that only valid data can be appended to the blockchain. It is the consensus mechanism that enables the gate-keeping function to be entrusted to a network of participants, rather than a single central authority.

Introducing Cryptocurrencies

Cryptocurrencies are the best known blockchain applications.

Cryptocurrencies are sent and received in transactions. These transactions must be processed and included in the blockchain by the corresponding cryptocurrency network, such as the Bitcoin or Ethereum networks, in a process known as mining. Each transaction must pay some ether, which is the cryptocurrency used by Ethereum, to the network to be processed. There is much debate about how cryptocurrencies should be valued and regulated, but the purpose of this course is to look at the practical uses of the new and powerful technology of blockchain. Concepts such as mining and sending and receiving payments will be covered later in this course.

Introducing Networks and Smart Contracts

Although the Bitcoin network is the most famous of the blockchain technologies, there are a number of other networks that focus on solving different problems from both inside and outside the blockchain sphere.

The following are examples of some blockchains:

- **Litecoin** is almost identical to the Bitcoin network, but has lower fees and is faster. It achieves this by having smaller "blocks", which are built faster.
- **The Ripple network** is run by a private institution that works with large companies to enable bank-to-bank transactions, and also has tokens for loyalty points and mobile minutes.
- **Ethereum** enables the implementation of **smart contracts**.

People tend to imagine that a "smart contract" is some kind of "electronic contract" which is "signed" between two Ethereum addresses. The reality is quite far from this. A **smart contract** is a (software) robot who controls an Ethereum address. It can operate at its controlled address the same way that a human user can operate at his/her address.

A smart contract on the Ethereum network can do the following:

- Receive, hold, and send Ether
- Receive, hold, and send tokens
- Execute functions from any other contract/robot
- Create any kind of transaction in the Ethereum blockchain

Conceptually, you can say that such a robot has, inside the Ethereum network, the same *rights* as any human user.

In this topic, we have discussed what blockchain is, some of the applications of blockchain, and the breadth of the blockchain sphere. We have also discussed the Ethereum network and introduced the idea of smart contracts. We will look at Ethereum and smart contracts in much more depth later on when we learn how to build a smart contract. Before that, we will look at the underlying concepts behind all blockchain technology.

Cryptology and Keys

One of the most fundamental ideas behind all end-to-end computer technology is security, and behind that is cryptology/cryptography.

Keys are the foundation of cryptography. They are strings of bits that are used by a cryptographic algorithm to transform plain text into cipher text or vice versa.

They are typically composed of letter and number sequences, and sometimes symbols. Using a key, you can encrypt (code) a message in such a way that it can only be decrypted (decoded) by someone who possesses the same key.

In blockchain technology, the most commonly used keys are asymmetric keys. These keys have been paired together, but are not identical. One key in the pair can be shared with everyone; this is called the public key. The other key in the pair is kept secret; this is called the private key.

The public key is also known as an address, and it is the one that you share with people. The private key is what allows you to control that addresses' funds.

Traditional Codes and Cryptography

For centuries, societies have used different methods to convey messages in secret or coded ways. This includes written code, gesture languages, and hand signs. One well-known example of a coded system is the international Morse alphabet, which was originally used to transmit messages through telegraphic systems. In this system, letters are replaced by dots and dashes to encode a message that can be sent over a simple electrical system:

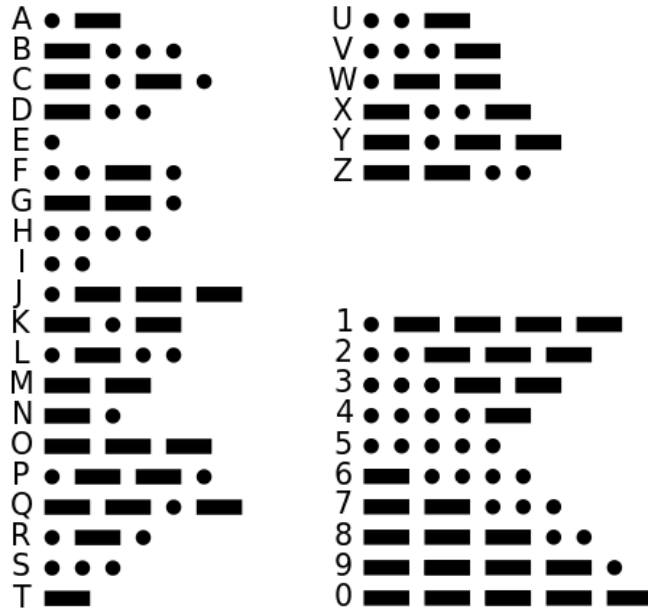


Figure 1.1: Morse code for letters and numerals

With Morse code, the key – meaning the information needed to encode and decode the message – is shared with all parties and is publicly available. Anyone who wants to read Morse code can easily get a copy of the key. Messages are encoded for ease of transmission.

With a cypher, the information is deliberately hidden from a third party by using a shared secret key that is only to be known by the intended senders and recipients. The key is often used with a cryptographic algorithm, which rearranges or substitutes letters in a message. Caesar's cipher is a good example of that:

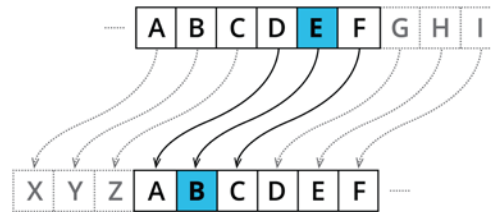


Figure 1.2: Caesar's cipher

In Caesar's cipher, which was used by the late Roman emperor, letters in the Latin alphabet were shifted three steps to the left. In this way, "HELLO WORLD" would become "EBIIL TLOIA."

In theory, someone with the key would be able to shift the letters back to reveal the message, while for an observer, the message would be nonsense. In practice, even an amateur codebreaker would have little difficulty in discovering the secret key:

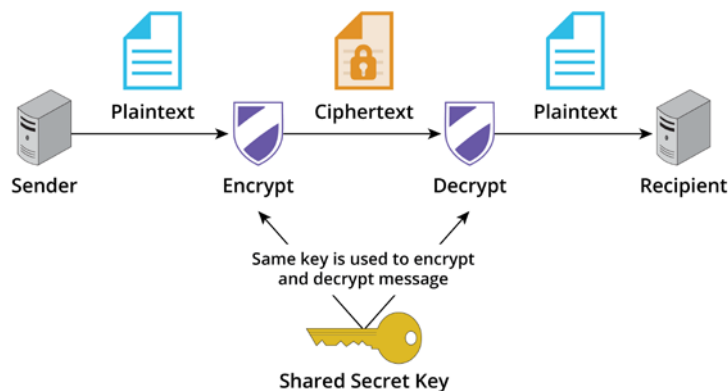


Figure 1.3: The process of encryption

The main characteristic of all traditional cryptographic systems is that you decode the message using the same key that you use to code it. Because the key is the same, this is known as symmetric cryptology.

The Enigma, used by Nazi Germany in World War II, is one of the most famous uses of symmetric but complex cryptography. It needed the sender and recipient to both be equipped with the same machines, which had to be set up identically. The machines worked as the algorithm, and the key would change every 24 hours. Famously, the Allies were able to intercept and decode these messages, and worked out a number of techniques for cracking the key daily.

The Allies had excellent cryptologists who were able to replicate the encryption method used by the Germans. This, together with the capture of key tables, hardware, and German procedural mistakes, is what led to a "crypto-victory" in the war that turned the tide in the Allies' favor.

New Cryptography

In 1974, two British mathematicians from **GCHQ (Government Communications Headquarters)** discovered a new way to implement encryption and decryption. They developed what is known as the RSA encryption algorithm.

In this new kind of system, two keys are used; one is used to encrypt the message, and the other is used to decrypt the message:



Figure 1.4: Symmetric encryption uses one key while asymmetric encryption uses two keys

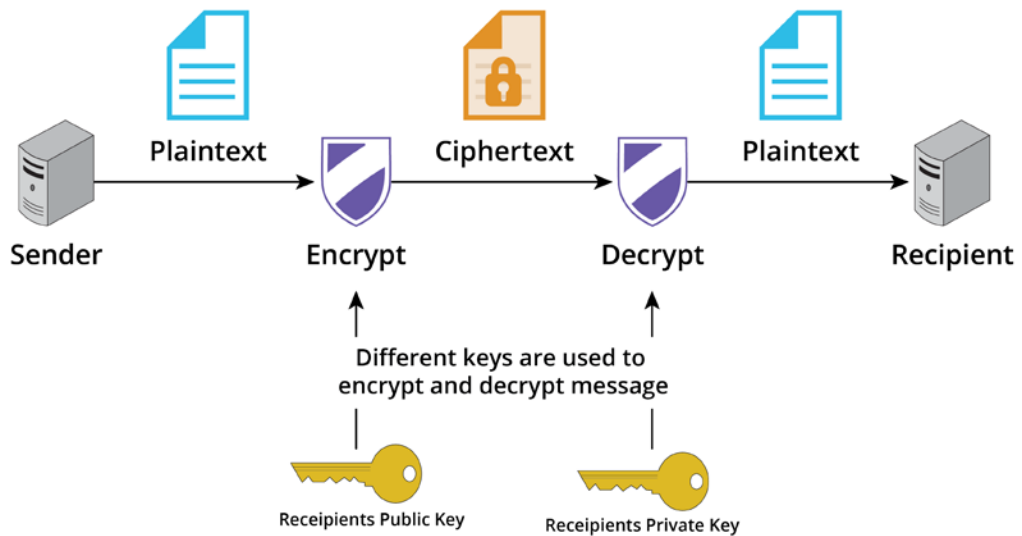


Figure 1.5: The process of RSA encryption

Originally, this development was only shared with the USA's CIA (Central Intelligence Agency). It remained a military secret until 1997, which is when the British government declassified the research.

This new kind of encryption is known as asymmetric encryption, because the key used to encrypt is different to the key used to decrypt.

Unlike what happens with symmetric encryption systems, where a secure channel is needed to transmit the encryption key, you can transmit a public key over insecure channels. This is because it uses complex algorithms, and there is no efficient solution to brute-force the finding of the private key. It does not matter how good a hacker is. It is estimated that it would take the whole mining power of Bitcoin about 653 million years to crack a single address.

In this topic, we have discussed how cryptology is key to blockchain technology, the different types of cryptology, and how blockchain uses public and private keys to send information. In the next topic, we will be looking at how to interact with the Ethereum network by using an account.

Opening an Ethereum Account

In this topic, we will look at how to open an Ethereum account, the difference between wallets and accounts, and how to use "toy" Ethereum money on the Rinkeby test network.

In order to interact with the Ethereum network, that is, to send and receive payments and deploy smart contracts, we first need to open an account. An Ethereum account is similar to a bank account, an accounting account, or a debit card account. It is sequence of numbers and letters that uniquely identifies all the operations that you perform on the Ethereum blockchain while using such an account.

In the Ethereum network, you may have to distinguish between three types of accounts, as follows:

1. Common accounts, which are controlled by a user, which is the same as in any other cryptocurrency.
2. Contract accounts, which are controlled by a software robot (known as a smart contract).
3. Multisign accounts, which are controlled by two or more users (to send/spend ether, two or more participants in such an account must approve of the transaction).

Account Numbers and their Associated Private Keys

As we mentioned earlier, when an account sends or receives ether, a permanent record is kept on the blockchain. As the processing of the transaction requires computation work, the sender must pay a fee to the processor. This is also true in the case of deploying smart contracts.

This payment of fees creates a problem while writing and testing cryptocurrency software, as many rounds of testing may be needed until a program works fine. To do this in the live main network would be way too expensive. The solution is to have multiple chains, one running with real ether (real money) and others running with test (toy) ether (not real money).

Network type and name	Means of Payment
Live or Main Network	(Using real Ether and so real money)
Rinkeby Test Network	(Toy Ether; we will be using such network in this course)
Kovan Test Network	(Toy Ether)
Ropsten Test Network	(Toy Ether)

Figure 1.6: Various network types and the payment modes

Wallets

In blockchain, a **wallet** holds the public and private keys that you use to add and read data to/from the blockchain. It can be thought of as the blockchain's version of a bank account. There are different types, including paper wallets, which as described, are simply pieces of paper with key details on. Software wallets can allow you to manage one or more accounts and will normally have the functionality to allow you to receive and send Ether. Many wallets are specific to the network they work upon. Most Ethereum wallets will also allow you to execute functions on contracts.

The following is a list of the different types of wallets that are available:

- **Offline** or **hardware** wallets are small devices that occasionally connect to the web to enact blockchain transactions, often through a USB connection on a computer. They are extremely secure, as they are generally offline and therefore not hackable.
- **Paper wallets** are perhaps the simplest of all the wallets, these are pieces of paper on which the private and public keys of a bitcoin address are printed.
- **Online wallets** offer increased convenience; you can generally access your bitcoin from any device if you have the right passwords. All are easy to set up; come with desktop and mobile apps, which make it easy to spend and receive bitcoin; and most are free. The disadvantage is lower security. With your private keys stored in the cloud, you have to trust the host's security measures, and that it won't disappear with your money, or close down and deny you access. Some leading online wallets are attached to exchanges. Some offer additional security features such as offline storage.
- **Desktop wallets** are the original bitcoin wallets that were used by the pioneers of the currency. Computers installed with these wallets form part of the core network and thus have access to all transactions on the blockchain.

BitcoinQt was the first ever bitcoin client wallet built. Many believe it is what Satoshi Nakamoto used. With it, you play a role in the overall state of the network. Another bitcoin client wallet with richer features is Electrum, which is a lightweight client.

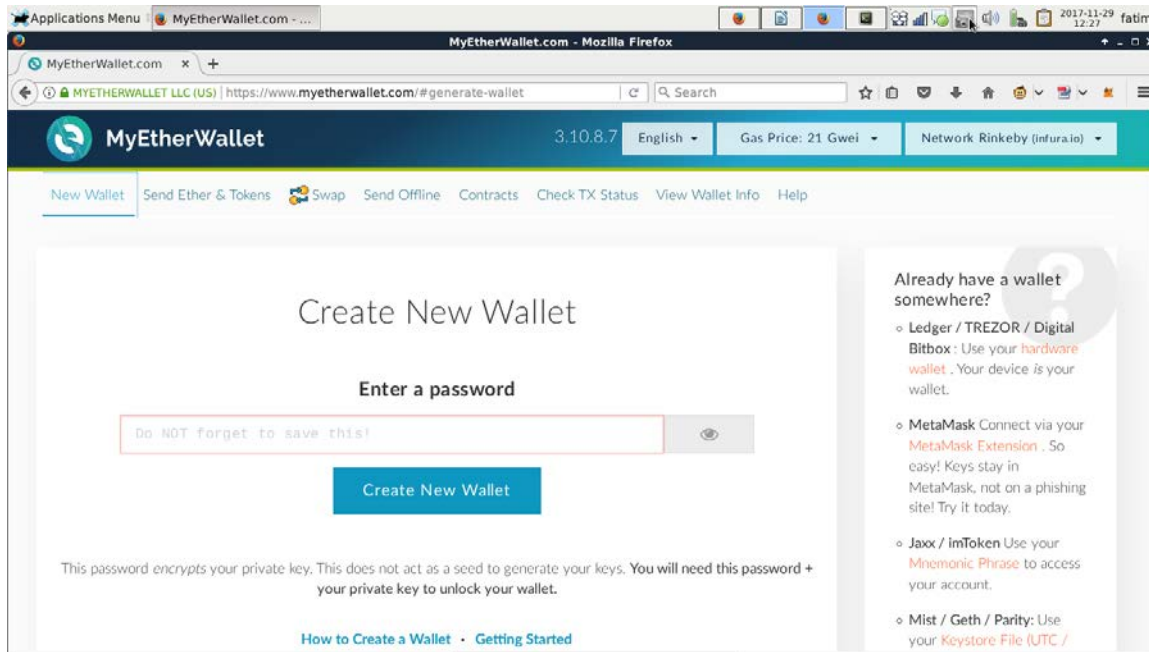


Figure 1.7: The MyEtherWallet home page

Cryptocurrencies work in a special way. To use a cryptocurrency, you need to use a wallet. While some people use the words "wallet" and "account" as synonyms, this is incorrect. Your wallet will contain your accounts. You either download one and install it, or you can use one online. After you have made your choice, the next step is to create an account.

We will be using myetherwallet.com in this course as it is quite popular and easy to use.

Exercise 1: Creating a Wallet and Safeguarding its Information

The wallet will need to hold test Ether and execute functions in contracts.

For this exercise, we will need a contemporary system with a current browser version:

1. Select the Rinkeby (etherscan.io) test network by using the upper-right dropdown menu.
2. Create a strong password. Save this in a safe place. It will be needed often throughout this course.

3. Enter your password and click on **create account**.
4. Carefully store the private key file. This will be needed often in this course.

After you create an account, you will get a screen similar to the following:

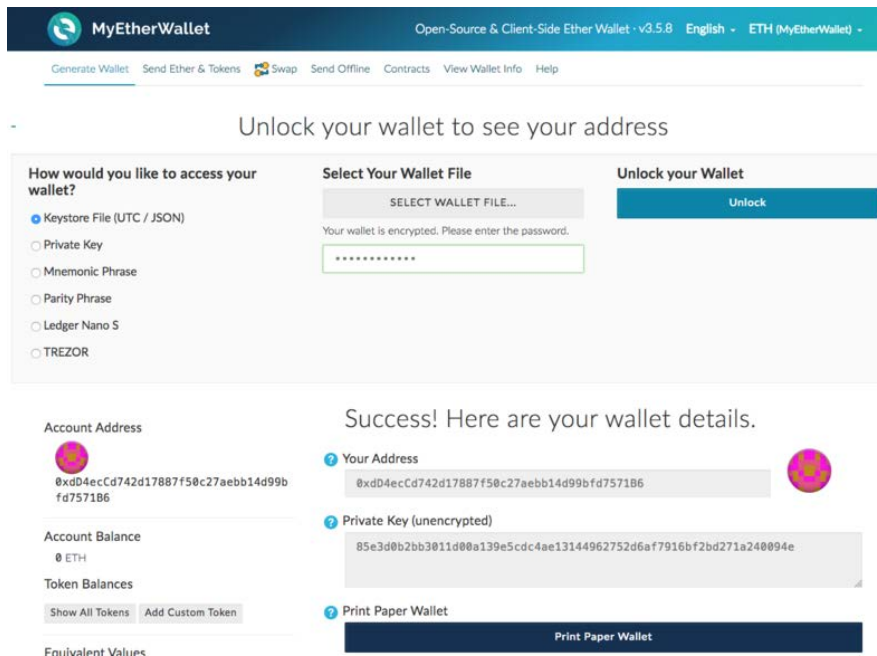


Figure 1.8: A screenshot of the EtherWallet

Private Keys and Public Keys

As we explained previously, every Ethereum account works by using both a private key for the owner to sign transactions, and a public key for everybody else to read such transactions.

Some things to note about the document are as follows:

1. Private Key

The format of your private key will be similar to the following:

3a1076bf45ab87712ad64ccb3b10217737f7faacbf2872e88fdd9a537d8fe266

2. Account or Address

The format of your account (which is generated from your public key) will be similar to the following:

0xC2D7CF95645D33006175B78989035C7c9061d3F9.

Note that there is a lowercase version of an address as follows:

0xc2d7cf95645d33006175b78989035c7c9061d3f9

A partially uppercase version of an address is as follows:

0xC2D7CF95645D33006175B78989035C7C9061D3F9.

The partially uppercase version has a checksum to verify the address.

Using your Wallet

Now that you have chosen a wallet and created at least one account inside it, you can use your account to receive and send transactions.

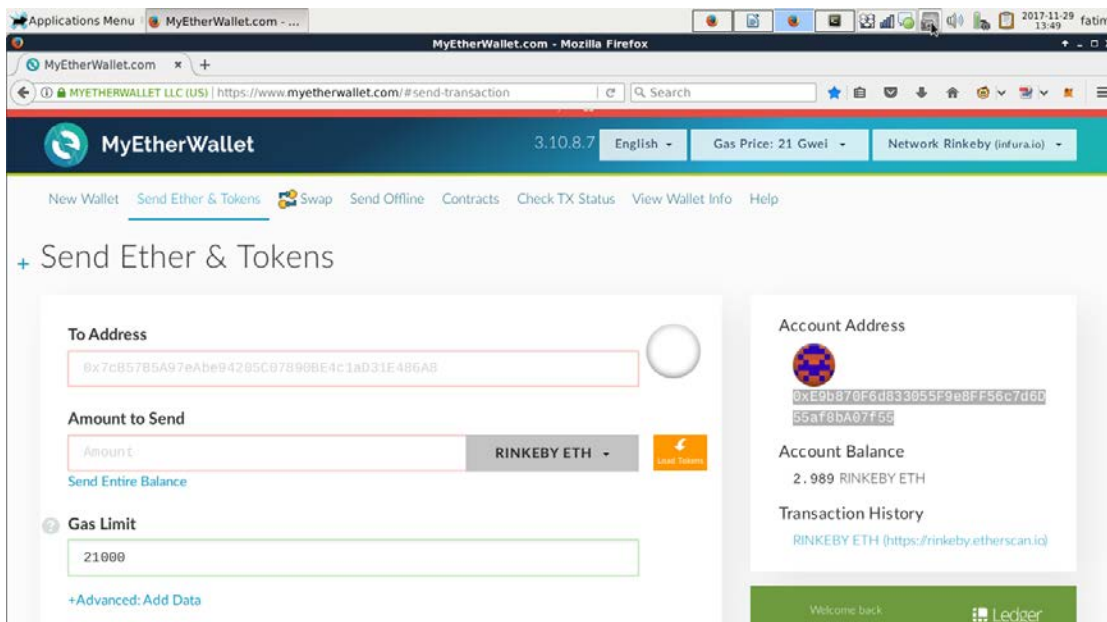


Figure 1.9: A screenshot of the EtherWallet

Exercise 2: Getting the Toy Ether from the Rinkeby Test Network

We will get the toy ether from the Rinkeby test network faucet, <https://faucet.rinkeby.io/>.

This toy ether is required to make transactions on the Rinkeby test network.

For this exercise, we need to have an Ethereum wallet:

1. To request funds via Twitter, send a tweet with your Ethereum address pasted into the contents (the surrounding text doesn't matter). Copy/paste the tweet's URL into the input box on the page and then click on **Give me Ether**.
2. Check the progress of your requests on the same page.

Note

To request funds via Google Plus, publish a new public post with your Ethereum address embedded into the content (the surrounding text doesn't matter). Copy/paste the post's URL into the input box on the page, and then click on Give me Ether. To request funds via Facebook, publish a new public post with your Ethereum address embedded into the content (the surrounding text doesn't matter). Copy/paste the following URL into the input box on the page, and then click on the button to provide Ether: <https://www.facebook.com/help/community/question/?id=282662498552845>

While barcodes are good for numbers, there is a kid on the block who can handle numbers and letters efficiently: **Quick Response codes**, or **QR codes** for short.

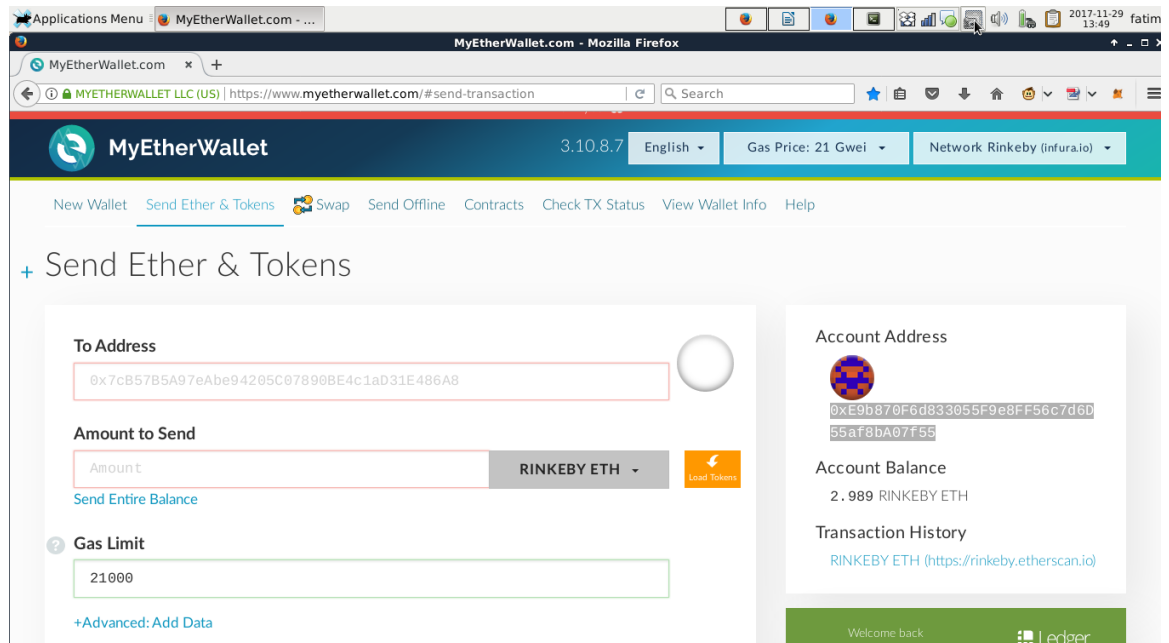


Figure 1.10: A screenshot of the EtherWallet

Addresses are sometimes shown as QR codes. This is practical because QR codes can be easily read by using a phone application.



Figure 1.11: Ethereum account/address (L) and Ethereum private key (R)

QR codes can be read by installing a QR code reader application, such as **Barcode Scanner**.

We have now learned how to set up accounts and wallets, how to use public and private keys and their associated QR codes, and how to use the Rikeby test network to send and receive toy ether. Next, we will take a deeper look at the Ethereum network, nodes, and mining.

The Ethereum Network, Nodes, and Mining

In this topic, we will be looking at the network of computers that underlies Ethereum, what a node is, and how mining works to keep the network running.

The Ethereum Network

There are many machines on the internet that use Ethereum. Collectively, we call them the Ethereum network. Some just hold a copy of the Ethereum blockchain, while some hold a copy and perform mining, approving Ethereum network transactions.

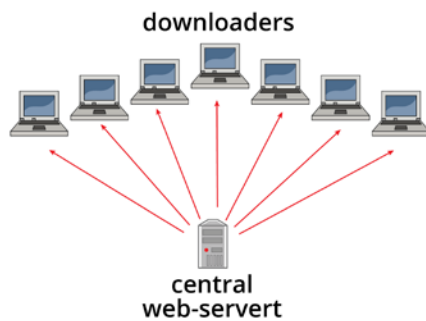
One of the most common mathematical models used to demonstrate the concept of fault tolerance is the **Byzantine Generals' Problem**.

The Byzantine Generals' Problem is an agreement problem in which a group of generals, each commanding a portion of the Byzantine army, encircle a city. These generals wish to formulate a plan for attacking the city. In its simplest form, the generals must only decide whether to attack or retreat. Some generals may prefer to attack, while others prefer to retreat. The important thing is that every general agrees on a common decision, for a halfhearted attack by a few generals would become a rout and be worse than a coordinated attack or a coordinated retreat.

The problem is complicated by the presence of traitorous generals who may not only cast a vote for a suboptimal strategy, but do so selectively. For instance, if nine generals are voting, four of whom support attacking while four others are in favor of retreating, the ninth general may send a vote of retreat to those generals in favor of retreat, and a vote of attack to the rest. Those who received a retreat vote from the ninth general will retreat, while the rest will attack (which may not go well for the attackers). The problem is complicated further by the generals being physically separated and having to send their votes via messengers who may fail to deliver votes or may forge false votes.

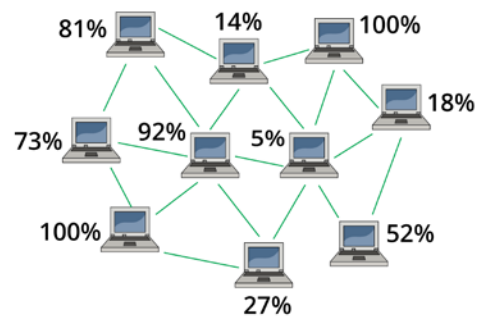
Byzantine fault tolerance can be achieved if the loyal (non-faulty) generals have a majority agreement on their strategy. The typical mapping of this story onto computer systems is that the computers are the generals and their digital communication system links are the messengers.

Traditional Centralized Downloading



- Slow
- Single point of failure
- High bandwidth usage for server

Decentralized Peer-to-Peer Downloading



- Fast
- No single point of failure
- All downloaders are also uploaders

Figure 1.12: The difference between centralized downloading and peer-to-peer downloading

In order to classify them, we can consider two broad cases:

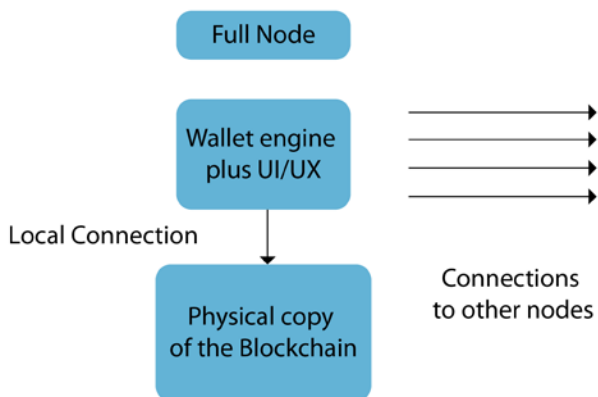


Figure 1.13: A diagram depicting the connection of the wallet to the physical copy of the blockchain

- If you use a web wallet, such as myetherwallet.com or any other similar service, you can submit transactions, but you *cannot* run any Ethereum software components
- If you run a full local wallet, such as Geth or Parity, which holds a copy of the Ethereum blockchain, then you are running what is called an Ethereum Node

Nodes

A machine that is running an Ethereum client, such as Geth or Parity, which holds a copy of the blockchain, is called an Ethereum node.

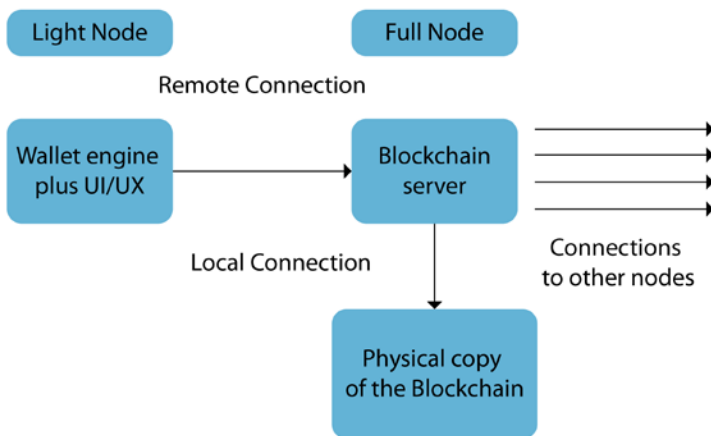


Figure 1.14: A diagram depicting a full node and a light node

So, when you run Ethereum software on your machine, you are running an Ethereum node. We can classify nodes into three groups:

- **Light nodes:** They only run a wallet; they do not locally store any of the Ethereum blockchains (be it live, Rinkeby, or Kovan). These nodes usually run MetaMask, Exodus, or a similar wallet.
- **Full nodes:** They run a wallet and store one full copy of one of the Ethereum blockchains locally (be it live, Rinkeby, or Kovan). These nodes usually run Parity, Geth, or a similar wallet.
- **Miners:** They not only store one full copy of one of the Ethereum blockchains locally, but they also receive transactions and group them to add new blocks to the blockchain that they hold a copy of. To do this, they run Ethereum mining software, for example, **Ethminer**.

Mining

This is an essential activity to keep the Ethereum network running. Roughly, it consists of the following tasks:

1. To get new transactions from other nodes.
2. To perform hashing work, usually in a team with other miners, in what is called a Mining Pool. This must be done until a new block is found. Such a block is added to the blockchain. All transactions that are included in such a block get their first confirmation.

A hash function is a mathematical process that takes input data of any size, performs an operation on it, and returns output data of a fixed size. For a new block to be considered valid, a hash needs to be found that, when converted to a number, will be equal or lower than a certain number. Taking into account this for a given output, it is not possible to calculate the input, and so finding a block is a very difficult task.

3. To pass the new blocks to other nodes. This activity extends the Ethereum blockchain by adding new blocks to it and is the only way in which Ethereum transactions can be approved. Each time a group of miners (also known as a **Mining Pool**) adds a new block to the blockchain, they get a reward in ether. Every new block adds one confirmation to the transactions of the previous blocks. New blocks contain a hash of the previous block. Because of this, if a previous block is changed – even a single letter – the hash will radically change.

Transactions and Blocks

In the previous topic, we looked at network, nodes, and mining. In this topic, we will look at how transactions are made, recorded, and passed on to the next blocks. We will also look at hashing, the concept of gas, and confirmations.

Transactions and Calls

While using the Ethereum blockchain (either live or on a test network) you can do two different things: you can issue transactions (which write data to the blockchain, and so spend "gas," which is equivalent to ether) or you can perform calls (which do not modify the blockchain, and so they are free).

Calls

You can query existing values in the blockchain for free. These may include the following instances:

- Checking the status of a transaction
- Reading a public variable from a contract
- Executing a function from a contract that does not modify variables, and so does not modify the blockchain

These actions do not generate transactions, only read existing transactions, and so don't require network fees or consume "gas."

Exercise 3: Calling the Ethereum Network

1. Go to <https://www.myetherwallet.com/#contracts>.
2. Under **Select Existing Contract**, you can find many contracts.
3. Select one contract and look for a call. We will use the **Athenian: Warrior for Battle** contract and call the **Total Supply** function.

Transactions, Transaction Hashes, and Gas

When you send ether, send tokens, create a contract, or modify one or more contract variables, you are issuing a transaction, and so you have to pay network fees. These fees are measured in "gas," which has a different equivalence to real or toy Ether in each Ethereum network.

Each transaction is (at least) composed of the following components:

- A sending address
- A receiving address
- Data
- A transaction hash, that is, a sequence of characters that is calculated on the basis of all previous values

Different blockchain networks offer different ways to check activity. Etherscan is one way to check the activity of all transactions for a user address on an Ethereum network, and is shown in the following screenshot (note the transaction hashes and gas consumption):

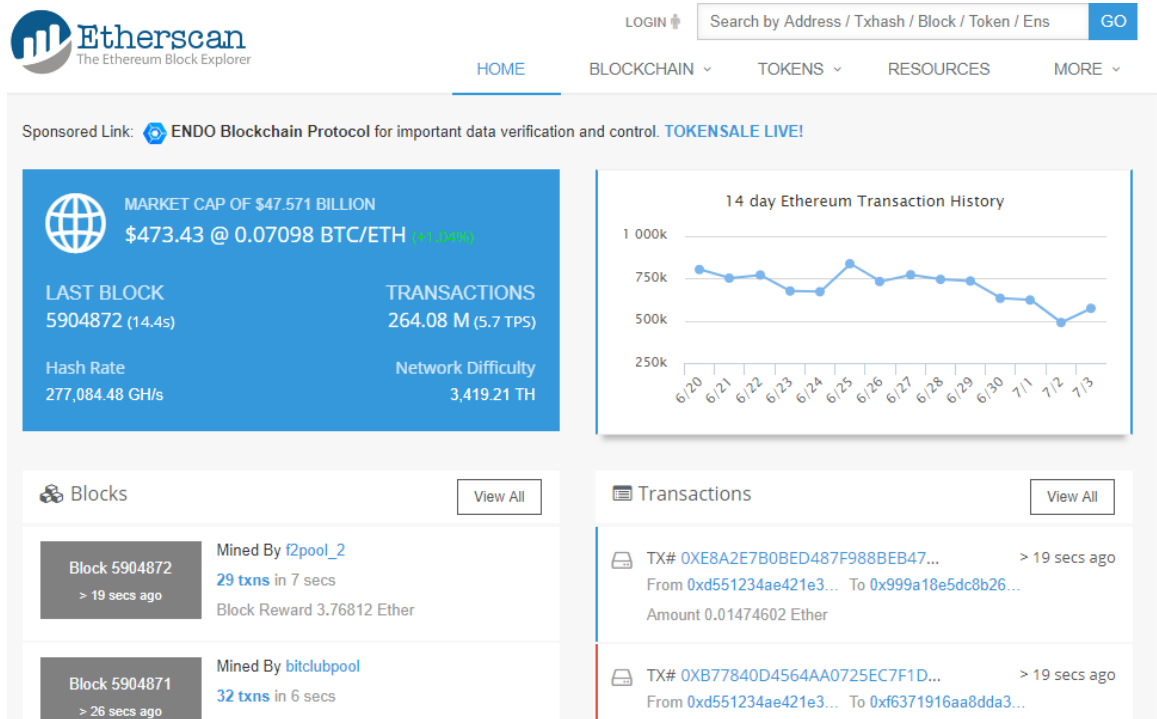


Figure 1.15: The Etherscan home page

It can also display all of the transactions for a smart contract, as shown in the following screenshots:

The screenshot shows the Etherscan interface for a smart contract. At the top, the contract address is `0x306E5D0C7b3934AF9BD57C3Ef0Eb886982C2aEE`. Below this, there's a sponsored link for Ubex.com. The main section is divided into 'Contract Overview' and 'Misc'. The 'Contract Overview' shows a balance of 0 Ether, an ether value of \$0, and 450 transactions. The 'Misc' section includes an 'Add To Watch List' button, the contract creator's address, and a 'View (\$0.00)' button for token balances. Below this is a navigation bar with tabs for 'Transactions', 'Internal Transactions', 'Token Transfers', 'Code', 'Read Contract', 'Write Contract', 'Events', and 'Comments'. The 'Transactions' tab is active, showing a list of the latest 25 transactions. The table below contains the following data:

TxHash	Block	Age	From	To	Value	[TxFee]
0xbd49bdd84a46c7...	4952944	163 days 13 mins ago	0x1c714f60af98b51...	0x306e5d0c7b3934...	0 Ether	0.0007973
0x558434b8d4a022f...	4947797	163 days 21 hrs ago	0x10edc948c6bc87...	0x306e5d0c7b3934...	20 Ether	0.000663575

Figure 1.16: Etherscan displaying all of the transactions for a smart contract



LOGIN

Search by Address / Txhash / Block / Token / Ens

GO

HOME

BLOCKCHAIN

TOKENS

RESOURCES

MORE

Transaction 0x36624b54f18b73aec3a19492dc2defae162490310a98508d7605ebcdaa153a18

Home / Transactions / Transaction Information

Sponsored Link: [Quadrant Protocol sold-out private sale, public Token Sale live. Enable a data driven world](#)

Overview	Event Logs (1)	Comments
Transaction Information		
Tools & Utilities		
TxHash:	0x36624b54f18b73aec3a19492dc2defae162490310a98508d7605ebcdaa153a18	
TxReceipt Status:	Success	
Block Height:	4661168 (1243683 block confirmations)	
TimeStamp:	214 days 7 hrs ago (Dec-02-2017 08:19:45 AM +UTC)	
From:	0x7e23e8fb2c971f6ce8b208dfcd79c7159ec8a37	
To:	Contract 0x58ca3065c0f24c7c96aee8d6056b5b5decf9c2f8	
Token Transferred:	From 0x7e23e8fb2c971f6... To 0x306e5d0c7b3934... for 398 ERC20 (GXC)	

Figure 1.17: Etherscan displaying details for sending tokens

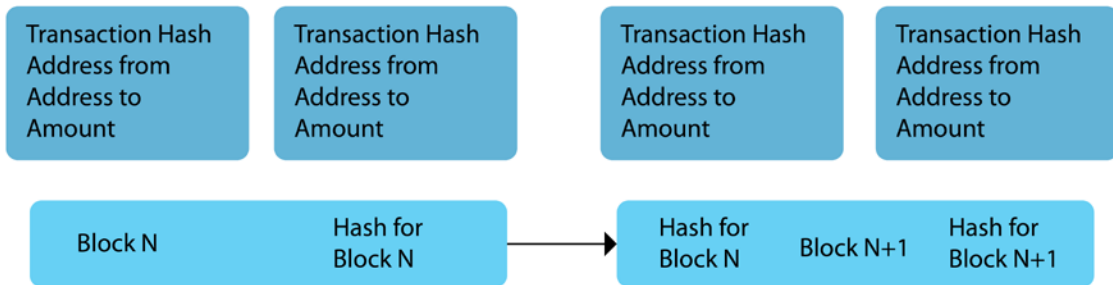
Note

The difference between calls and transactions is that transactions are recorded on the blockchain, whereas calls are not. Essentially, calls only work locally in a contract, and do not broadcast to the blockchain, and thus don't cost any gas. Transactions are broadcast, and if mined will impact the blockchain.

Blocks and Block Hashes

Transaction data is permanently recorded in files called blocks. They can be thought of as transaction ledgers. Blocks are organized into a linear sequence over time, called a block chain, and each block has a corresponding hash.

Blockchain can be fairly compared to a general ledger. In accounting, this is a book that contains all transactions for an institution. While the book is composed of pages and the blockchain of blocks, conceptually they are very similar.



Physical copy of the Blockchain

Figure 1.18: The physical copy of the blockchain

When transactions are received by mining nodes, they enter a queue, and when they are processed, they are grouped in blocks. A block contains at least the following:

1. A hash for the previous block (to form the blockchain)
2. Transactions, each one structured as described in *Subtopic C*
3. A hash for the current block

So, an exception has been made for the first block, which is also called the "genesis" block; each block B contains its own hash, plus the hash for block B-1. This is one of the characteristics that makes blockchain technology unique.

The following screenshot provides an example of a block:

Block Information	
Height:	Prev 4691765 Next
TimeStamp:	209 days 16 hrs ago (Dec-07-2017 04:20:02 PM +UTC)
Transactions:	86 transactions and 24 contract internal transactions in this block
Hash:	0x4df39d77a9f26d52a277a40e4345a4f9fab7cb9c4ba77c4cf7eb01e4475d888a
Parent Hash:	0x6f226c4f2cd34e93c94b62cefad40ce5b2316d1e6f3b9bec171ab4106c13184a
Sha3Uncles:	0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
Mined By:	0xb2930b35844a230f00e51431acae96fe543a0347 (miningpoolhub_1) in 12 secs
Difficulty:	1,485,367,850,042,680
Total Difficulty:	1,670,220,998,354,936,026,407
Size:	16052 bytes
Gas Used:	6,689,877 (99.76%)
Gas Limit:	6,705,806
Nonce:	0xd655b6d80c38995e
Block Reward:	3.358837478802142439 Ether (3 + 0.358837478802142439)
Uncles Reward:	0
Extra Data:	t10 (Hex:0x743130)

Figure 1.19: A screenshot providing information of a block.

Confirmations

Each time a new block is found and is added to one of the Ethereum blockchains (live, Rinkeby, or Kovan) by miners, all the transactions included in it are confirmed. When you check one of them on one of the Ethereum blockchain explorers, you will see that it shows one confirmation. Let's call this block B.

When the next block is mined, all of the transactions included in it will also get confirmed and show one confirmation. Let's call this new block B+1. When B+1 is mined and all of its transactions get one confirmation, all transactions in block B get two confirmations.

Then, block B+2 gets mined and all of its transactions get one confirmation. Transactions in block B+1 get two confirmations, transactions in block B get three confirmations, and so on. You probably already see the pattern here.

Transactions can be checked in the following Ethereum blockchain explorers:

- Ethereum blockchain Explorer and Search: <https://etherscan.io/>.
- Home: The Ethereum blockchain explorer: <https://etherchain.org/>.
- Ethplorer: Ethereum token explorer and data viewer: ethplorer.io/.
- Ethereum Classic Block Explorer | GasTracker.io (Ethereum Classic: gastracker.io/).

In this topic, we have looked at key concepts, including the difference between transactions and calls, transaction hashes, blocks and block hashes, gas, and confirmations.

In the next topic, we will look at how to send and receive transactions and check their statuses.

Sending and Checking Transactions

Having looked at what a blockchain is, how it is recorded, and the network and key concepts such as hashing and cryptography, it is now time to start sending and receiving transactions.

Sending Transactions

To send a transaction, you need the following components:

- A wallet
- An address inside that wallet (most wallets will let you create more than one address)
- An Ether balance in any of your addresses
- A recipient's address
- An internet connection to broadcast your transaction to the other nodes in the Ethereum network, until it reaches a miner and is mined

Caution

Many public Wi-Fi networks, for security reasons, block TCP ports other than 80 (the one used for the World Wide Web), so even if you are able to visit websites, your wallet may be unable to send transactions.

Exercise 4: Sending and Receiving Transactions

We need a contemporary system with a current browser version to do this exercise. We also require a wallet that's already been created with some toy Ether in it. Before starting this exercise, students should swap wallet addresses either by email or by generating a QR code:

1. In the wallet, go to **Send Ether and Tokens**.
2. Open a private key file.
3. Enter a password.
4. Enter the receiver address in the **To Address** field.
5. Enter the amount to send.
6. Wait for the gas limit to be calculated.
7. Click on **generate transaction**, click **send transaction**, and then click **yes**. Finally, click on check **tx status**.

Receiving Transactions

It is also important to be able to receive transactions as well as send them. To receive a transaction, you need a wallet and an address inside that wallet (most wallets will let you create more than one address) so that you can share your public address with the sender.

Checking Transactions

Once you get a notification that a transaction has been sent to one of your addresses, you can check the validity of your transaction in two ways:

- Wait until your wallet receives the transaction
- Ask the sender for the transaction hash that can be entered in the "search" field of any blockchain explorer service

Summary

In this lesson, you have discovered the basics of the Ethereum blockchain. You should now understand the basics of modern cryptography and the difference between symmetric and asymmetric cryptology. You now have basic knowledge of the Ethereum network and how to work with transactions using blockchain.

It is important to remember that blockchain is not just about the ups and downs of the cryptography market, rather that it is a new paradigm in information technology with a myriad applications, one of which is the concept of smart contracts. We will look at smart contracts in more depth in the next lesson, *Lesson 2, Smart Contracts and Solidity Language*, and start building our first one.

2

Learning Solidity

Learning Objectives

By the end of this lesson, you will be able to:

- Describe the basic framework of the Solidity language
- Use the Ethereum blockchain and the Ethereum network
- Write a smart contract in Solidity
- Compile, deploy, and test smart contracts in the Rinkeby test network

In this lesson, we will examine the Solidity language that will be used to build our distributed Apps. Then will write a program enabling us to deploy a token on the blockchain.

Introduction

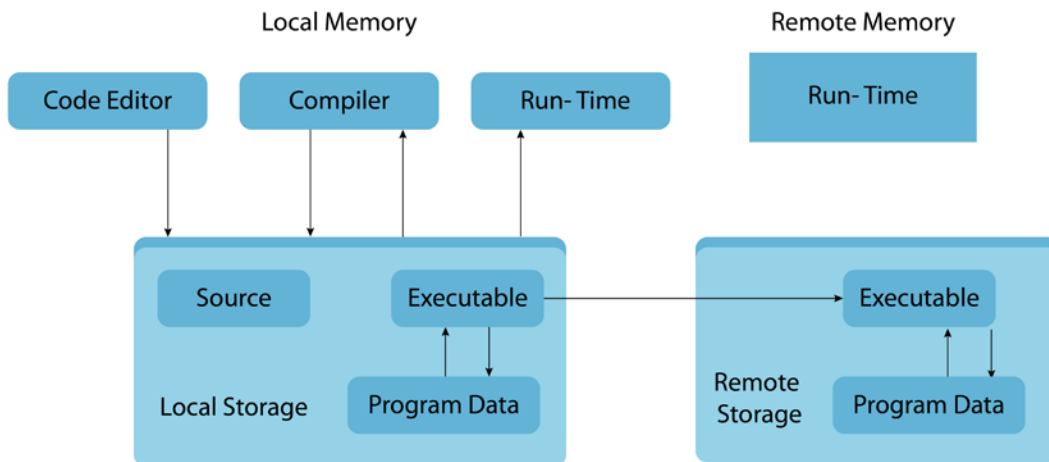
In the previous lesson, we learned the fundamentals of the Ethereum blockchain, including pivotal concepts on keys and cryptology, Ethereum accounts, network nodes and mining, blockchain, and how to send transactions.

The Solidity Language

Solidity is a high-level language that was specifically designed for writing smart contracts. Its syntax may remind you of popular contemporary languages such as Python, C++, and JavaScript.

Solidity is statically typed and supports inheritance, libraries, and complex user-defined types, among other features.

Using Solidity will open up to you a completely new programming model. You will learn by example by creating an ERC20/ERC223 token by means of a Smart Contract.



Traditional Programming System

Figure 2.1: A traditional programming system

Until recently, for most developers, coding was made up of basic steps such as creating source code, compiling it, and then running it.

All three of these steps usually happen on the same computer while creating software and, after that, any number of people can use the created program on their own computers. However, due to the distributed nature of blockchain, building computer code for networks has a number of extra steps.

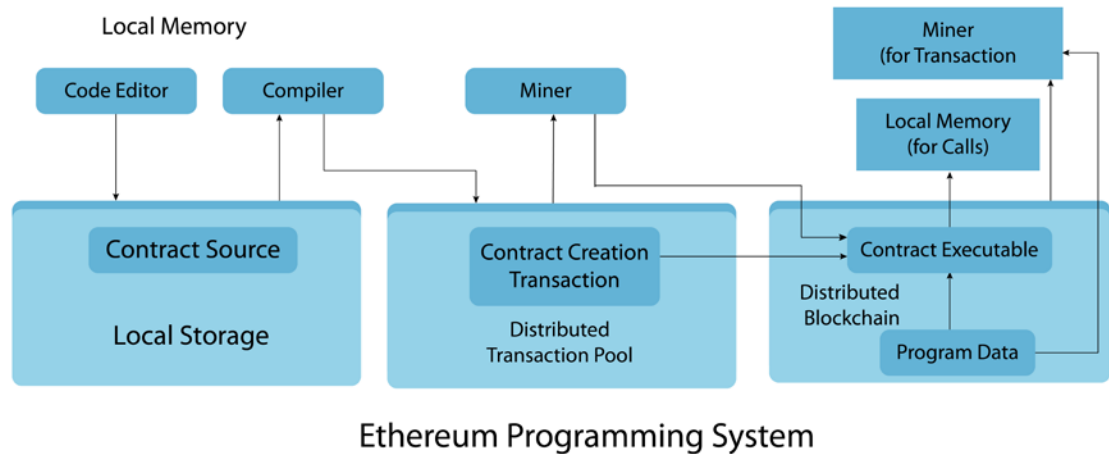


Figure 2.2: An Ethereum programming system

In 2015, Ethereum brought us the first practical blockchain-based distributed processing model, and with it a new programming paradigm.

When using the Ethereum network, programming has an increased number of steps:

1. The source code is created
2. The code is compiled
3. The compiled code is deployed to the blockchain by means of an Ethereum transaction
4. The transaction is taken by a miner and put into a block
5. Calls are made to the program, also known as the **smart contract**, to read variables
6. Transactions are issued to the contract to modify variables

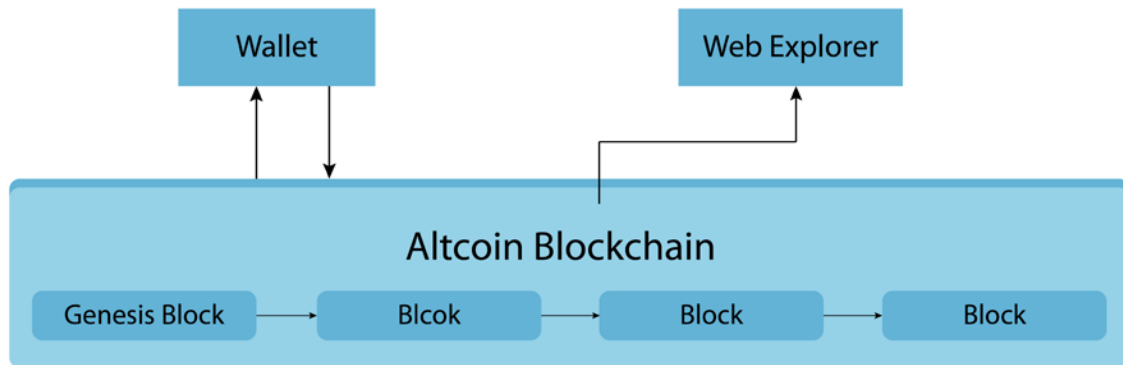
Now, we are going to explore putting this new programming model into practice.

Your First Smart Contract

In this topic, we will be creating our first smart contract. Previously, we learned that smart contracts are robots that control Ethereum addresses.

We learned that they can:

- Receive, hold, and send Ether
- Receive, hold, and send tokens
- Execute functions from any other contract/robot
- Broadcast transactions to the Ethereum blockchain, for example, a transaction that calls a function that changes a contract's owner



Infrastructure for an Altcoin

Figure 2.3: The infrastructure of an Altcoin

One of the basic things you can do with a blockchain is create your own cryptocurrency. Creating a fully-fledged cryptocurrency (that is, an Altcoin) means implementing the following points:

- A wallet, capable of coin operations such as writing transactions and reading transactions, and also capable of mining to create new blocks
- A blockchain with a genesis block, that is, the first block in the blockchain
- A block explorer to display blockchain transactions

In the context of cryptocurrencies, a wallet is a software program that allows you to send and receive coins. Technically speaking, a wallet stores private and public keys and interacts with the network.

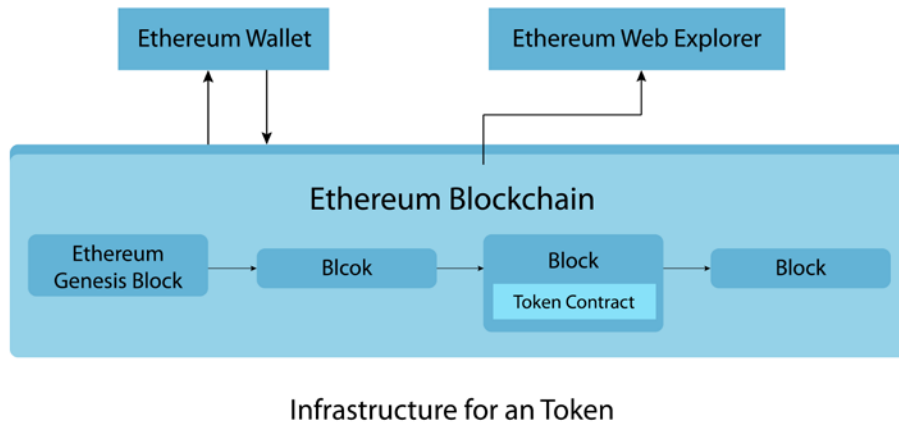


Figure 2.4: The infrastructure of a token

Using smart contracts, you can create what is known as a **token**: a cryptocurrency that runs on Ethereum's infrastructure. Your token will be able to operate using a standard Ethereum wallet, record transactions on Ethereum's blockchain, and be visible in Ethereum's block explorers.

Some Ethereum wallets will perform token operations for you, so it will write token transactions to the Ethereum blockchain and will also be capable of reading them. Transactions related to your token will be recorded in the Ethereum blockchain, and you will need Ether to pay for recording them. Ethereum block explorers will take care of displaying transactions related to your token. This means that you will not need to create a fully-fledged infrastructure.

In the Ethereum network, you pay for computation. This is measured using gas. For every operation that a smart contract can perform, there is a specific cost, for example 6 gas or 30 gas.

Each unit of gas also has a price known as "gas price". It is set in gwei and directly translates gas to ether. It is important to note that gas is not a currency by itself, and only a measure of computational effort.

A Simple Token

```
contract MyToken {  
  
    /* Contract data: array with balances and initial number of tokens */  
    mapping (address => uint256) public balanceOf;  
    uint initialSupply = 1000000 public;  
  
    /* Initializes contract with initial supply tokens to the creator of the contract */  
    function MyToken() {  
        balanceOf[msg.sender] = initialSupply;           // Give creator all initial tokens  
    }  
  
    /* Send coins */  
    function transfer(address _to, uint256 _value) {  
        require(balanceOf[msg.sender] >= _value);       // Check if the sender has enough  
        require(balanceOf[_to] + _value >= balanceOf[_to]); // Check for overflows  
        balanceOf[msg.sender] -= _value;                 // Subtract from the sender  
        balanceOf[_to] += _value;                         // Add the same to the recipient  
    }  
}
```

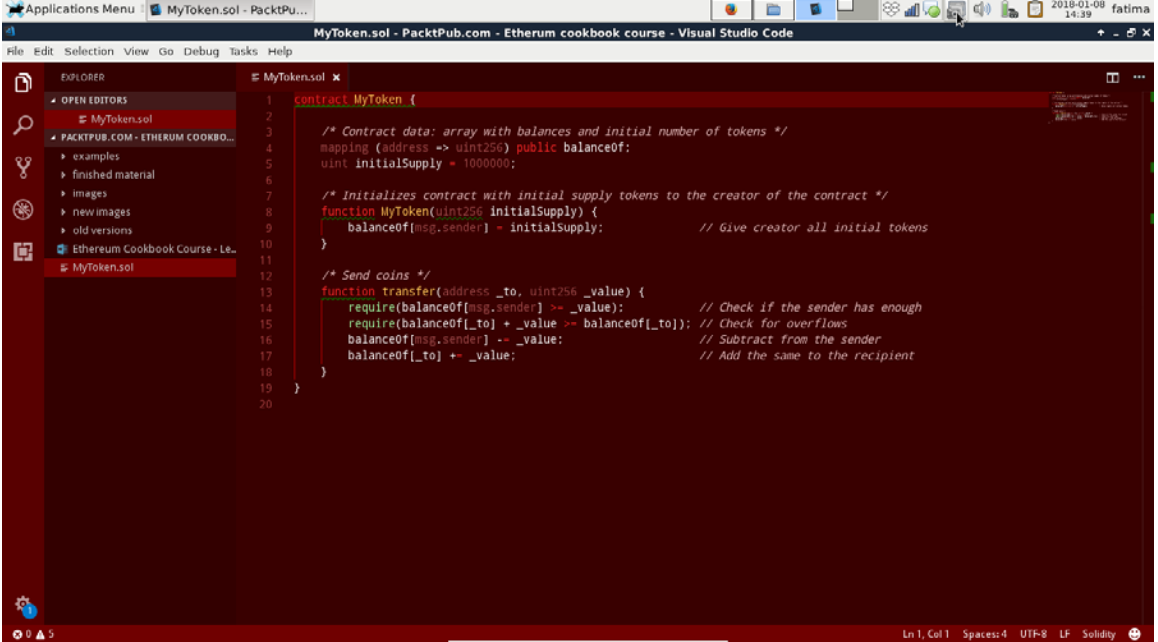
Figure 2.5: A simple token

The following are the functions for different sections of the Solidity source code:

- Contract name
- Contract data (including the initial supply and a mapping to hold balances)
- A constructor to initialize the contract during creation
- Functions to perform contract operations

Activity 1: Creating an Ethereum Token

Ethereum startups will, most of the time, want to create their own cryptocurrency, that is, an Ethereum token. We will create one here. Such a token can be used to implement discount coupons, mileage systems, and any kind of similar value-holding souvenir:



```

1  contract MyToken {
2
3      /* Contract data: array with balances and initial number of tokens */
4      mapping (address => uint256) public balanceOf;
5      uint initialSupply = 10000000;
6
7      /* Initializes contract with initial supply tokens to the creator of the contract */
8      function MyToken(uint256 initialSupply) {
9          balanceOf[msg.sender] = initialSupply; // Give creator all initial tokens
10     }
11
12     /* Send coins */
13     function transfer(address _to, uint256 _value) {
14         require(balanceOf[msg.sender] >= _value); // Check if the sender has enough
15         require(balanceOf[_to] + _value <= balanceOf[_to]); // Check for overflows
16         balanceOf[msg.sender] -= _value; // Subtract from the sender
17         balanceOf[_to] += _value; // Add the same to the recipient
18     }
19 }
20

```

Figure 2.6: The code for creating a simple token

We'll require a contemporary system with a current version of Visual Studio Code for this exercise. Our aim is to create a simple token based on the Ethereum network.

Let's perform the following steps to implement this activity:

1. Open Visual Studio Code.
2. Create a new file.
3. Insert the following code:

File name: Lesson 2_Activity 1.sol

```

contract MyToken {
    /* Contract data: array with balances and initial number of tokens */
    mapping (address => uint256) public balanceOf;
    uint initialSupply = 10000000 public;
    /* Initializes contract with initial supply to creator*/
    function MyToken() {

```

```
    balanceOf[msg.sender] = initialSupply;
    /*Above line gives creator all initial tokens*/
  }

  /* Send coins */
  function transfer(address _to, uint256 _value) {
    //[...]
  }
```

Live link: <https://bit.ly/2Ijq1ET>

4. Review the code for typos.
5. Save the file as **MyToken.sol**.

In this activity, almost everything happens around the **balanceOf** variable. Initially, it is declared using the following code:

```
mapping (address => uint256) public balanceOf
```

When the contract is initialized, the variable is also initialized by the following code:

```
balanceOf[msg.sender] = initialSupply
```

When making a transfer, the contracts checks funds, as shown in the following code:

```
require(balanceOf[msg.sender] >= _value);
require(balanceOf[_to] + _value >= balanceOf[_to]);
```

Finally, we make changes to the balance:

```
balanceOf[msg.sender] -= _value;
balanceOf[_to] += _value;
```

The whole contract is set out as follows:

```
contract MyToken {
  /* Contract data: array with balances and initial number of tokens */
  mapping (address => uint256) public balanceOf;
  uint initialSupply = 1000000 public;
  //[...]
  /* Send coins */
  function transfer(address _to, uint256 _value) {
    // Check if the sender has enough
```

```

    require(balanceOf[msg.sender] >= _value);
// Check for overflows
    require(balanceOf[_to] + _value >= balanceOf[_to]);
// Subtract from the sender
    balanceOf[msg.sender] -= _value;
}

```

Exercise 5: Using Remix to Compile Our Token

As we explained previously, Remix is a browser-based Solidity IDE. Among its features are the compiling, deploying, and debugging of smart contracts.

In the following exercise, we are going to use Remix to compile our token. In order to perform this exercise successfully, you will need a contemporary system with a current browser installed. This is very useful for situations where you need to check that your code is well-written:

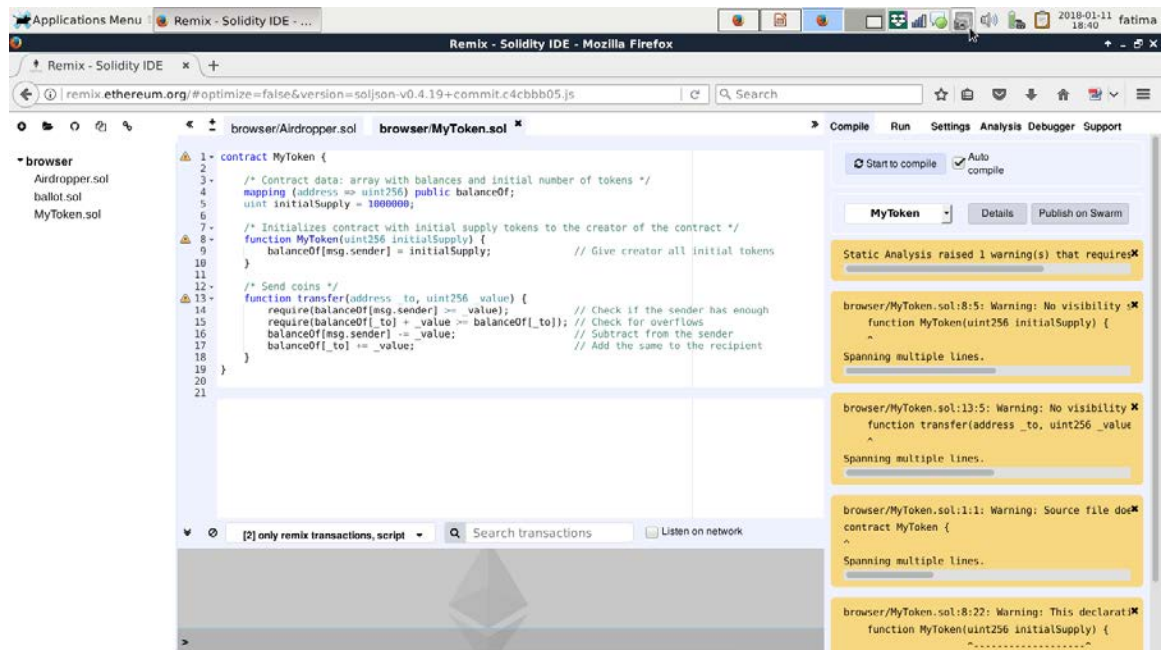


Figure 2.7: MyToken.sol

We'll complete the following steps to complete this exercise:

1. Open your web browser.
2. Go to remix.ethereum.org.
3. Create a new file using the plus sign on the upper left corner.
4. Go to Visual Studio Code and copy the preceding code to the clipboard.
5. Go to remix and paste the code in the newly created file.
6. On the right, if **Auto compile** is off, click on **Start to compile**.

Note

You can fix any compilation errors (no red errors, just orange warnings).

In this topic, we learned what a token is and created our own. In the following topic, we will go deeper into the Solidity language.

Basic Solidity

In this topic, we will learn about Solidity data types, variable scopes, collections, and mappings.

Solidity, as a programming language, can be considered one of the descendants of the Java language from the 90s. Its syntax may remind you of the syntax of the Java language, but also that of other descendants, such as JavaScript, Python, and PHP (if you have seen source code written in those languages).

Solidity Data Types

Solidity has Boolean, Integer, and String data types, which are similar to common programming languages' data types:

Data Type	Value
Boolean	True or False.
Integer	Signed and unsigned integers of various sizes. Keywords uint8 to uint256 in steps of 8 (unsigned of 8 up to 256 bits) and int8 to int256. uint and int are aliases for uint256 and int256, respectively.
String	As usual
Address	Specific for Ethereum addresses, holds a 20 byte value (size of an Ethereum address). Address types also have members and serve as a base for all contracts.

Figure 2.8: Solidity data types

Solidity also has a data type that is specific to the Ethereum blockchain environment. This is the address datatype, which defines a memory space for a valid Ethereum blockchain address.

Global and Local Variables

As a (sort of) Java descendant, Solidity allows you to define global variables, which is done at the start of the contract, and local variables, which are defined inside functions.

Note

Global variables, also called state variables, are permanently stored in a contract's storage. Local variables are created temporarily to hold values in calculating or processing something.

For the following example, at the beginning of the contract called **Variables**, we define a global variable called **globalVariable**. Then, we set its value with the **GlobalVariable** function and use the **getGlobalVariable** function to get its value. This value remains stored in the blockchain, so the next time the contract runs, you can retrieve its value again.

The contract also has a function called **getLocalVariable**, where a local variable is initialized and then returned. After the contract finishes executing, this local, temporary variable won't have a value anymore, as shown in the following code:

```
contract Variables {
    uint globalVariable;
    function setGlobalVariable(uint _global){
        globalVariable = _global;
        //[...]
    }
    function getGlobalVariable() constant returns(uint) {
        return globalVariable;
    }
    function getLocalVariable() constant returns(uint) {
        uint localVariable = globalVariable * 2;
        return localVariable;
    }
}
```

Collections

Solidity includes some of the usual collection types that are found in modern languages, such as Enum, Array, and Struct:

Collection Type	Value
Enum	A list of arbitrary values
Array	Indexed like in any language
Struct	Grouping of fields, similar to C structures
Mapping	Similar to a hash table

Figure 2.9: Collection types

The following are examples of how to declare collections:

- **Enum:**

```
enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
```
- **Array:**

```
uint[] anArrayOfNumbers = new uint[](7);
```
- **Struct:**

```
struct Campaign {
    address beneficiary;
    uint fundingGoal;
    uint numFunders;
    uint amount;
    mapping (uint => Funder) funders;
}
```

Mappings

Solidity also has a special type of collection called a mapping, which is particularly good for managing addresses. It is similar to a hash table (found in many modern languages), and it is good for managing address-value pairs.

In the following screenshot, you can see the declaration and contents of a mapping, holding addresses in the left column and the balance for each address in the right column:

The screenshot shows a Solidity code editor with the following code:

```
mapping (address => uint256) public balanceOf;
```

Below the code is a table representing the data stored in the mapping:

0x2dada28c49c7060ec2099b76b603909964148791	3420
0x2e3b8f4a9a1b7e371d0029a7a626bdf000a4f292	4000
0x3968755435de43e94174e8d123987ef3b3b267f9	3335
0x6d9da320dde848d9a82a134d6165acab7c811d0d	4421

Figure 2.10: Data types

Exercise 6: Creating Our Own Collection

Now that we have looked at the basics of Solidity, we are going to create our own collection for storing the details of each transaction. In the code that follows, a collection has already been created. Modify it to include more fields, specifically origin, target, amount, and the new balance for both addresses (called **balanceFrom** and **balanceTo**). It is very important that all fields are correctly stored in the transfer function.

The aim of this exercise is to create a collection that stores transaction details.

There are two sections to modify. First, the struct is defined. Second, the transaction details are stored in the struct. To finish, check that the code compiles. For this exercise to run successfully, we need a contemporary system with a current browser installed.

Let's perform the following steps:

1. Open Remix.
2. Create a new file.
3. Copy the original code.
4. Make the changes.
5. Check that the following code compiles:

```
contract MyToken {  
  
    /* Contract data: array with balances and initial number of tokens */  
    mapping (address => uint256) public balanceOf;  
    uint initialSupply = 1000000;  
    //[...]  
    uint amount;  
    //[...]  
    transfers.push(_td);  
}
```

The following is the solution (struct definition) for this exercise:

```
struct transferData{  
    address origin;  
    //[...]  
    _td.balanceTo = balanceOf[_to]  
    _td.balanceFrom = balanceOf[msg.sender]  
}
```


In this topic, we have looked at the basics of Solidity programming, as well as learned about the different data types, and the difference between global and local variables. We have also created our first collection. In the following topic, we are going to deploy and test a smart contract.

Testing Solidity

In the previous topic, we learned the basics of the Solidity language. We learned about the different data types (Boolean, integer, string, and address), about the different variable scopes (global and local), and the different collection types (enum, array, struct, and mapping).

In this topic, we are going to learn about the different Ethereum blockchains (Mainnet, Rinkeby, Kovan, and so on), the deployment process, and finally we will deploy a smart contract using Remix and MetaMask.

As in any language, once you get a piece of code that compiles, you need to test it to check that it does what it is supposed to do.

The Ethereum networks keeps a productive blockchain which uses real (and therefore valuable) Ether, and three test networks, which use toy (and so not valuable) Ether:

Type of Network	Means of Payment
Live or Main Network	(Using real Ether and so real money)
Rinkeby Test Network	(Toy Ether; we will be using such network in this course)
Kovan Test Network	(Toy Ether)
Ropsten Test Network	(Toy Ether)

Figure 2.11: A table depicting the different types of network and the available means of payment

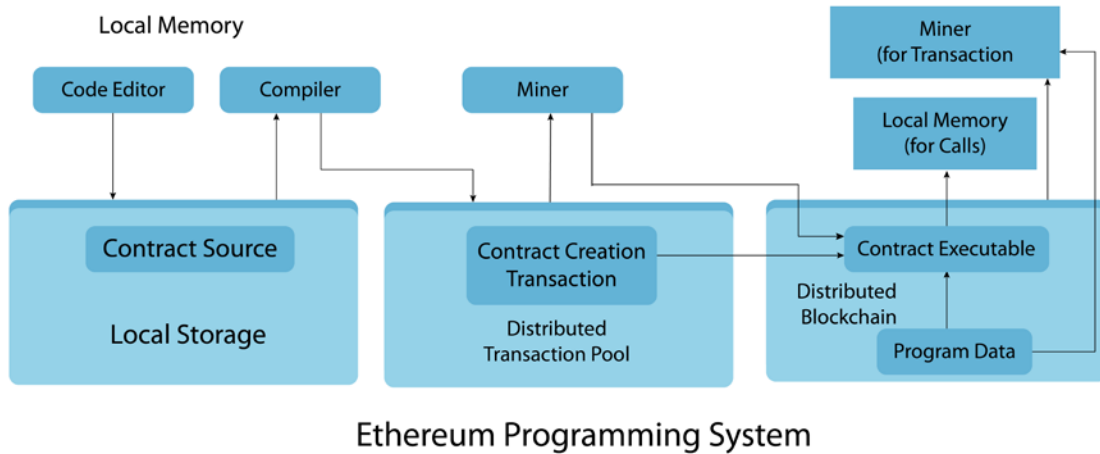


Figure 2.12: Ethereum programming system

To deploy a smart contract, you need to create the source file with a text editor, and then pass the text file through a compiler. Then, you must use a deployer (such as Remix) to assemble a contract creation transaction and send it to the Ethereum network. Once a miner processes your transaction and puts it into a block, you get an address for the contract.

Exercise 7: Deploying and Testing a Smart Contract

MetaMask is a browser extension that acts as a wallet as well as a bridge, allowing dApps to connect to the Ethereum blockchain:

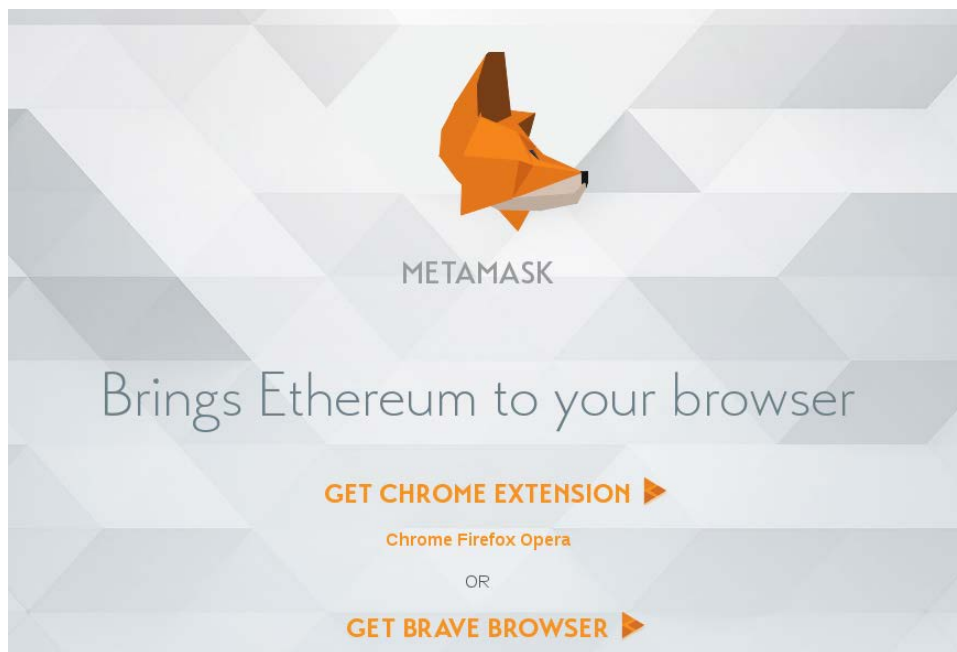


Figure 2.13: The MetaMask home page

To allow Remix to send transactions to the Ethereum network, you will need to install MetaMask.

The aim of this exercise is to deploy and test a smart contract using MetaMask, Remix, and MyEtherWallet. To complete this exercise successfully, you will need a contemporary system with a current browser installed (Chrome or Chromium).

Let's perform the following steps:

1. Open a web browser (Chrome or Chromium).
2. Go to metamask.io.

3. Click on **GET CHROME EXTENSION:**

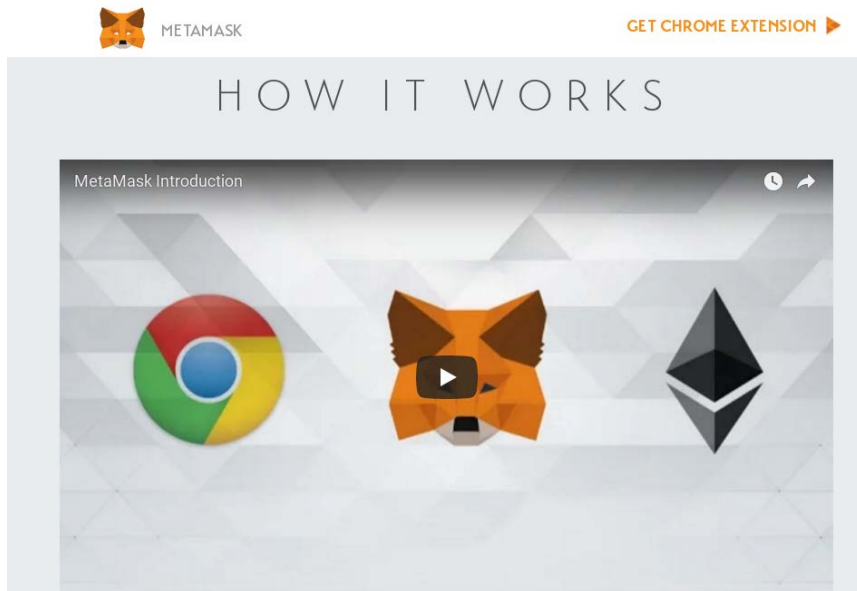


Figure 2.14: A How it works page containing a video, Introduction to MetaMask

4. In your browser, click on the fox head in the upper-right-hand corner, scroll to the bottom, and then click on **Accept:**

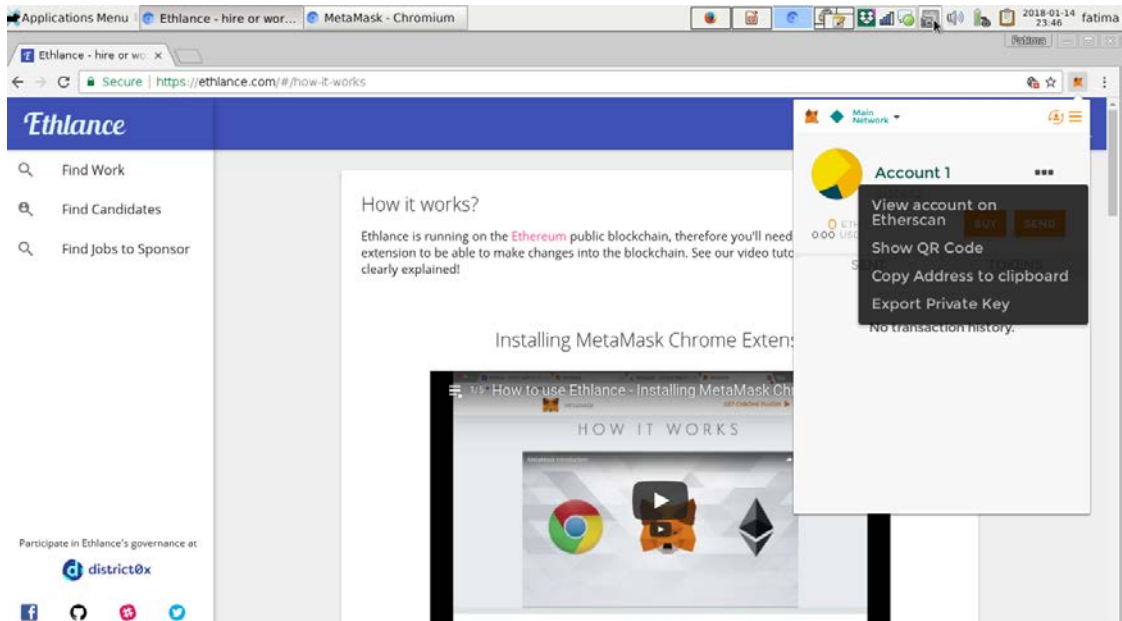


Figure 2.15: A screenshot showing the pop-up panel on the right

- Click on the three dots that are shown in the dropdown in the preceding screenshot and click on **Copy Address to clipboard**:

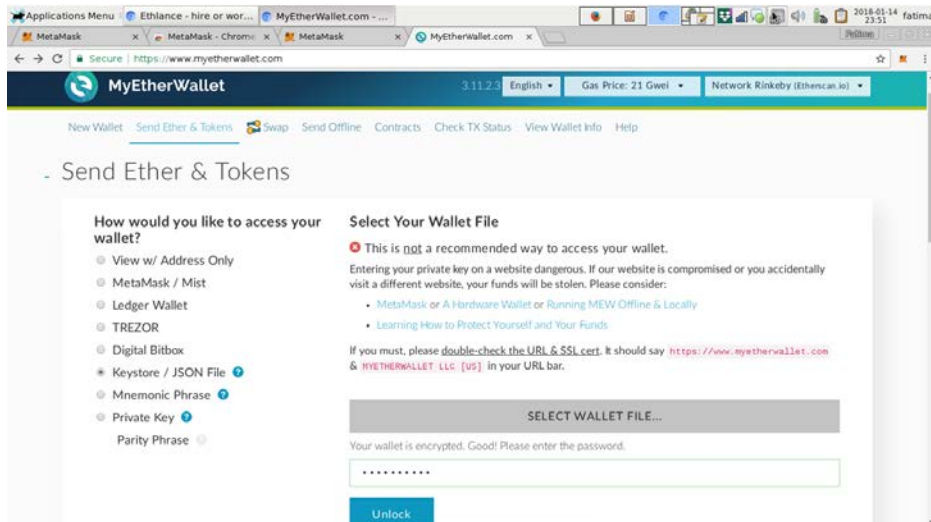


Figure 2.16: A screenshot of the Send Ether & Tokens page

- Go to MyEtherWallet.com.
- Click on **Send Ether & Tokens**.
- Click on **SELECT WALLET FILE....** This is the wallet file from Lesson 1, *Ethereum Blockchain*.
- Enter your password and click **Unlock**; you should get the following page:

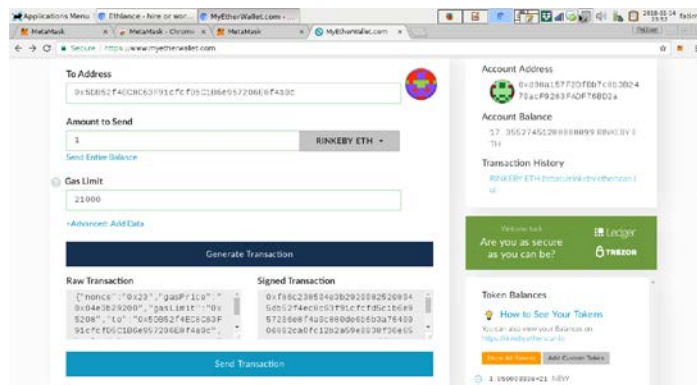


Figure 2.17: A page displaying To address, Amount to Send, and Gas limit

- In the **To Address** field, paste your MetaMask address.
- In **Amount to Send**, enter 1.

12. Wait for **Gas Limit** to fill itself. If it doesn't, enter **30000**.
13. Click on **Generate Transaction**.
14. Wait for the transaction to appear, and then click on **Send Transaction**:

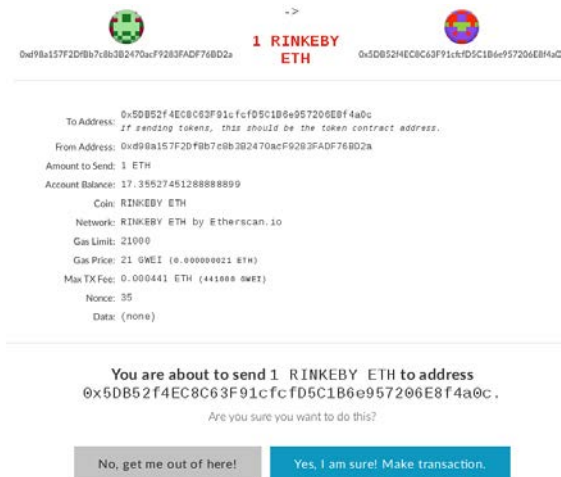


Figure 2.18: A page that shows the summary of the transaction and the confirms your decision

15. Click on **Yes, I am sure! Make transaction**.
16. Wait for the credit to appear in MetaMask.
17. Go to Remix.
18. Click on **Run**:

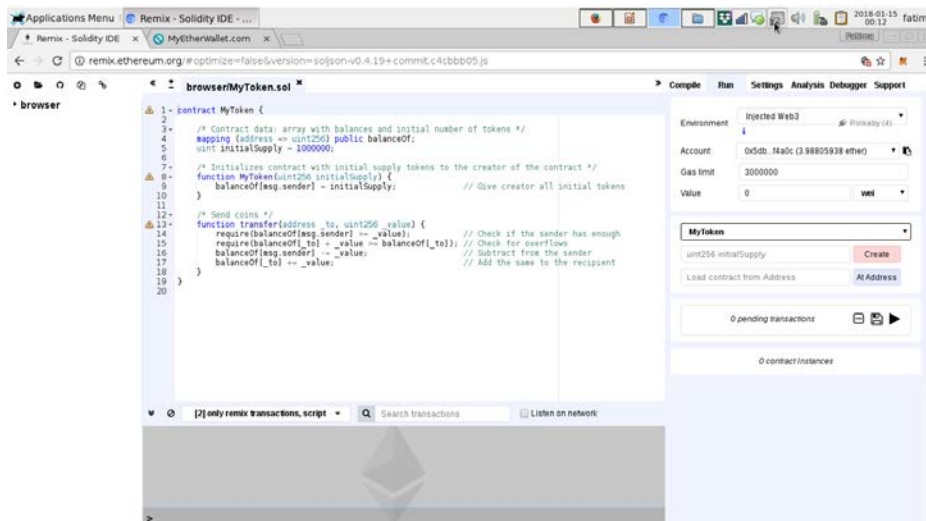


Figure 2.19: The Remix IDE with the code

19. In the MetaMask window, click on **SUBMIT**:

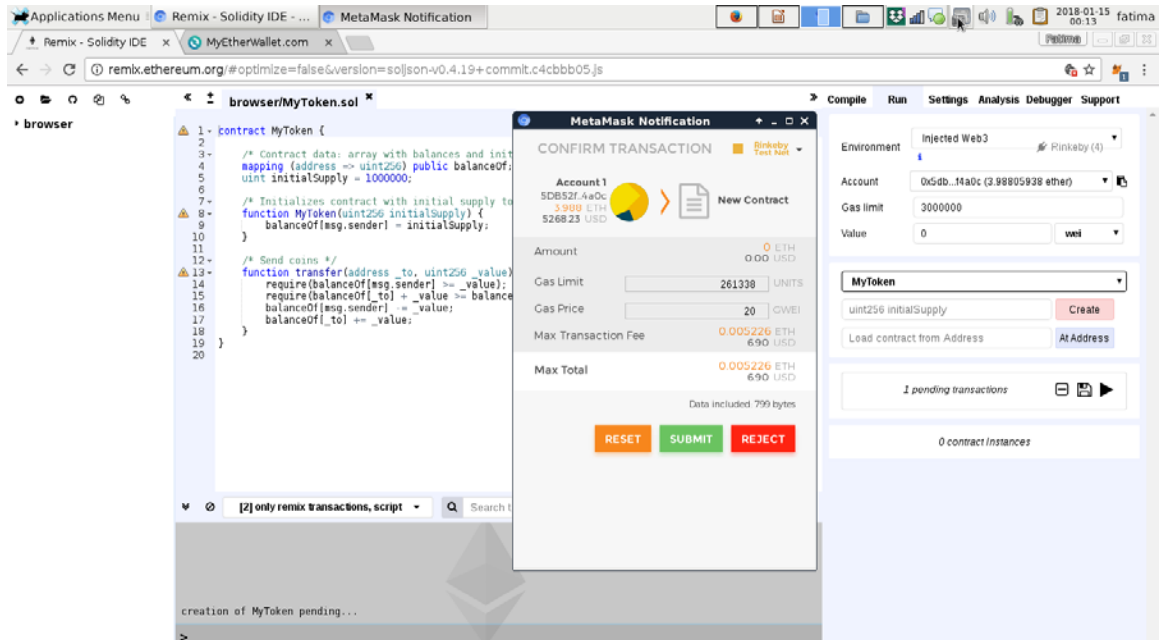


Figure 2.20: A screenshot of the MetaMask window

20. Wait for messages to appear at the bottom of the screen:

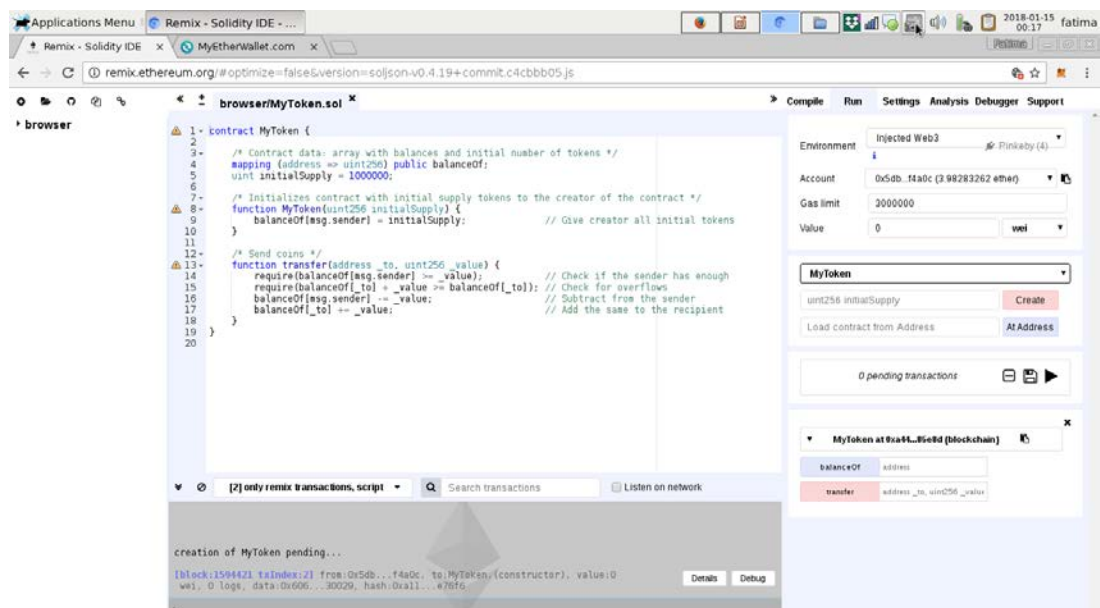


Figure 2.21: The Remix IDE

21. From the right-hand panel, copy the contract address to the clipboard:

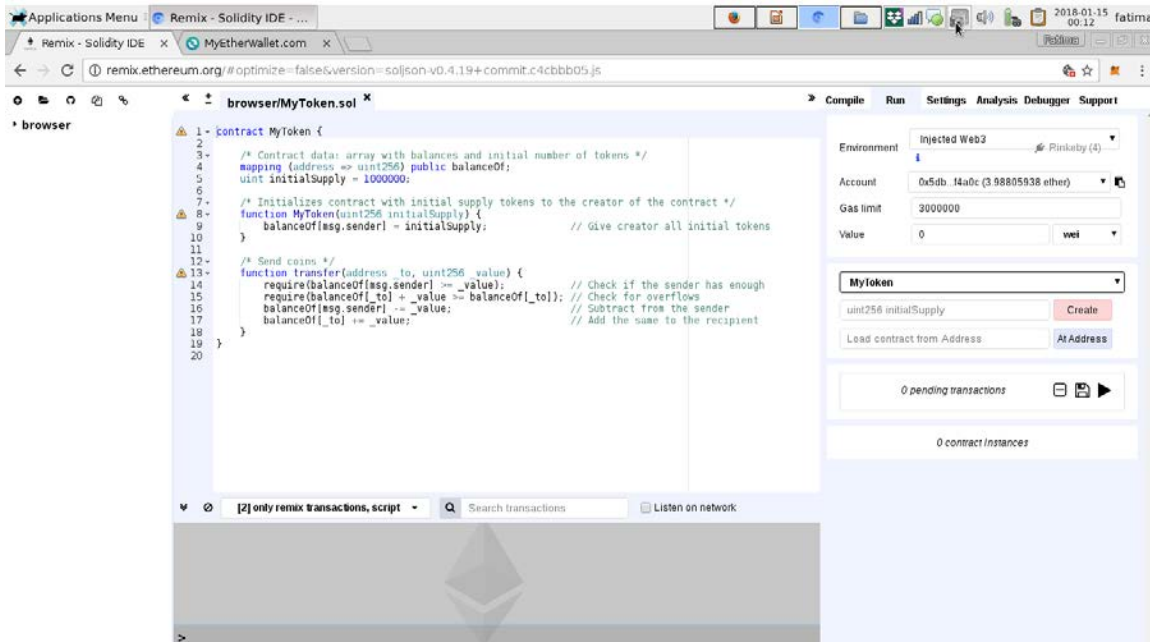


Figure 2.22: A screenshot of the panel that allows displays the contract address to be copied

22. Then, go to MyEtherWallet.com.

23. In **Contract Address**, paste your contract address:



Figure 2.23: A screenshot of the details of the Token

24. Go to Remix.
25. Click on **Details** on the lower-middle section of the page.
26. On the screen that follows, scroll down to **ABI**:



```
ABI    
▶ 0:  
▶ 1:  
▶ 2:  
  
WEB3DEPLOY    
  
var initialSupply = /* var of type uint256 here */ ;  
var mytokenContract = web3.eth.contract({'constant':true,'inputs':{'name':'','type':"  
var mytoken = mytokenContract.new(  
  initialSupply,  
  {  
    from: web3.eth.accounts[0],  
    data: '0x6060604052620f4240600155341561001657600080fd5b60405160208061031f83398101f  
    gas: '4700000'  
  }  
), function (e, contract){  
  console.log(e, contract);  
  if (typeof contract.address !== 'undefined') {  
    console.log('Contract mined! address: ' + contract.address + ' transactionHash'  
  }  
})  
}}  
◀  ▶  
  
METADATAHASH    
*db7f30e10052c438eabfd36c53cef9ff58cece219239d2e86c92666484f90763*
```

Figure 2.24: A screenshot of the ABI

27. Copy the ABI.
28. Then, go to MyEtherWallet.

29. In the **ABI / JSON Interface** field, paste your contract ABI:

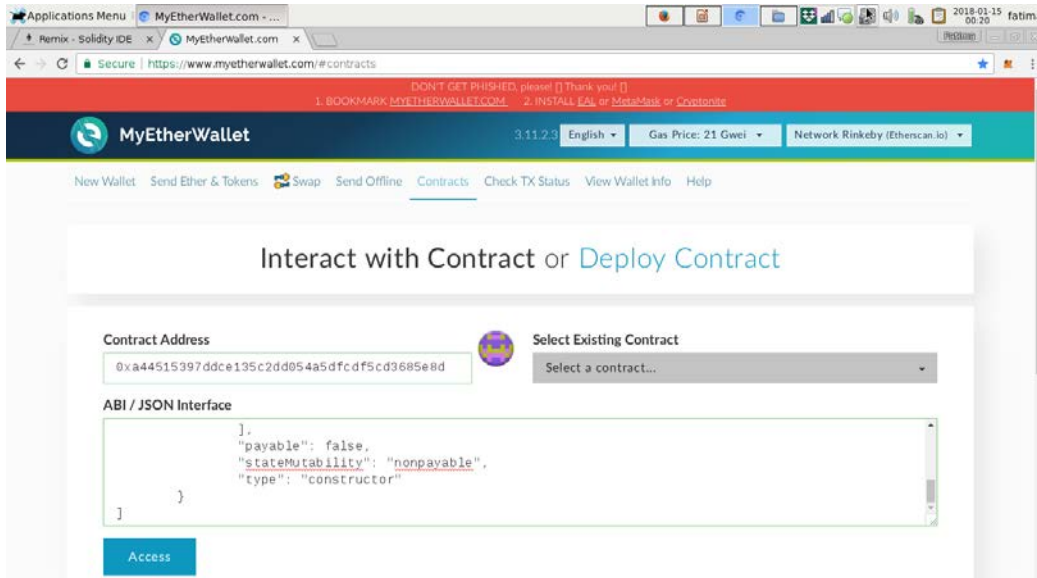


Figure 2.25: A screenshot of the Contract in the wallet

30. Click on **Access**.

31. Once **Select a function** becomes available, click on **initialSupply**:

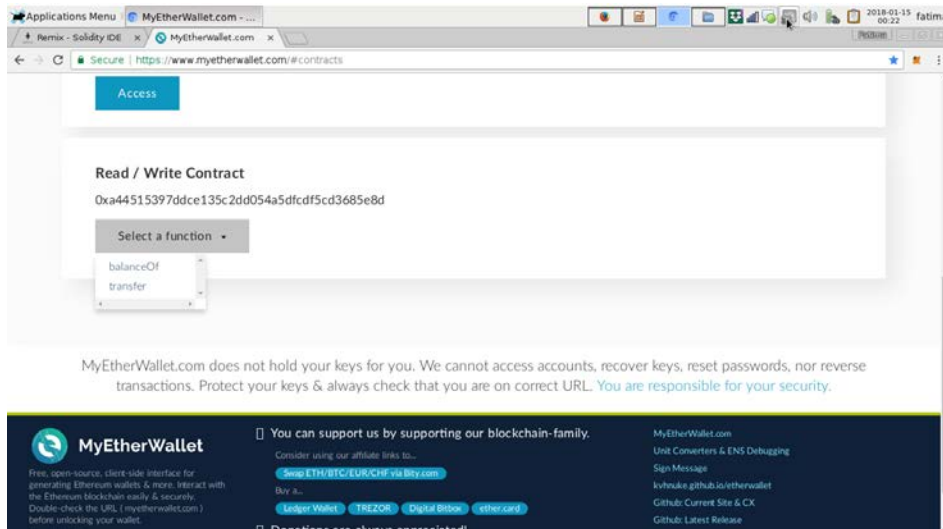


Figure 2.26: A screenshot that displays the deactivated Select a function button

32. Check your value.

33. Click on **balanceOf**:

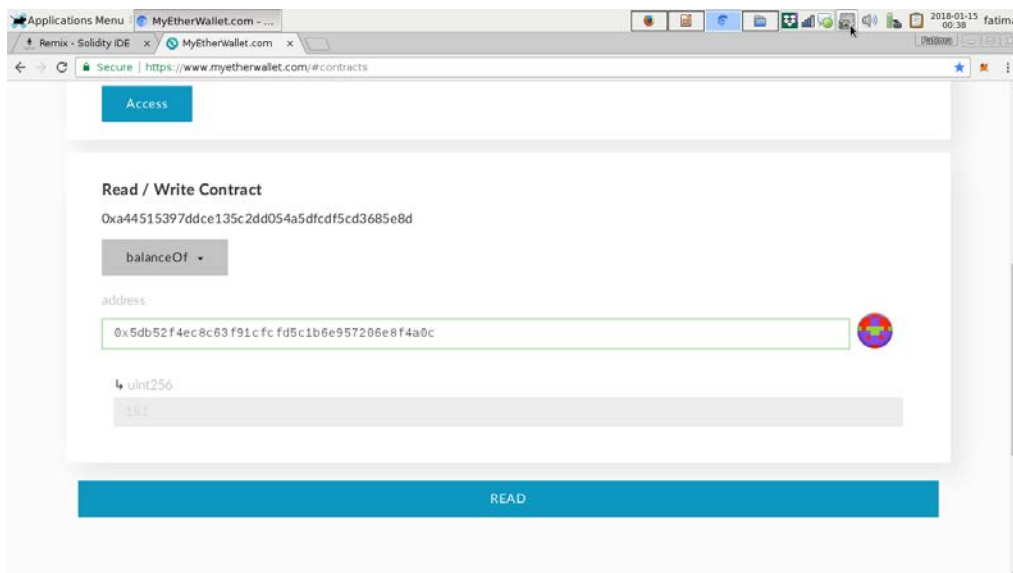


Figure 2.27: A screenshot with the balanceOf button that allows you to read the balance

34. In the **address** field, enter your MetaMask address.

35. Click on **READ**.

36. Check that your balance is equal to your initial supply:

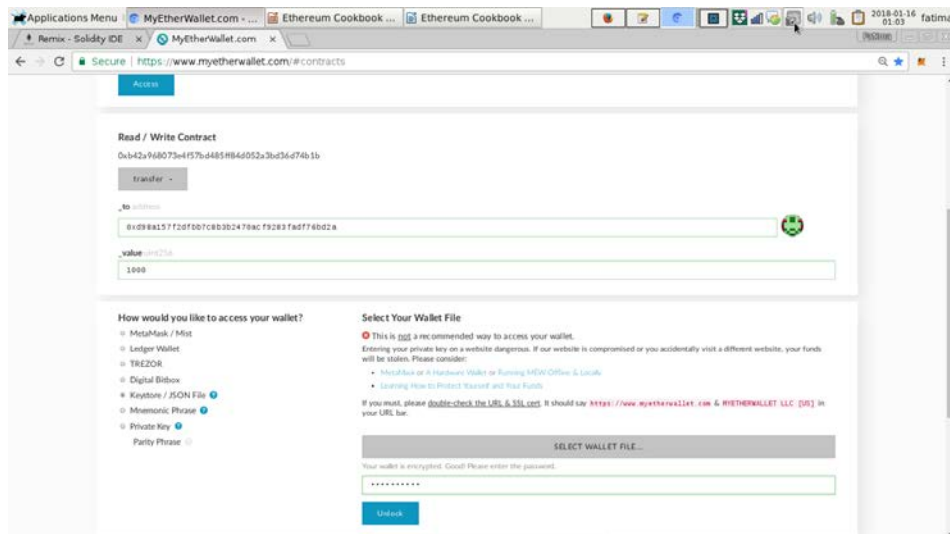


Figure 2.28: A screenshot displaying options to access the wallet

37. Click on **transfer**.

38. In the **address** field, enter your MyEtherWallet address.

39. In the **value** field, enter a random quantity from 1 to half of your initial supply.
40. Click on **Keystore / JSON file**.
41. Click on **SELECT WALLET FILE** and select your wallet file.
42. Enter your wallet's password.
43. Click on **Unlock**.
44. Click on **WRITE**:

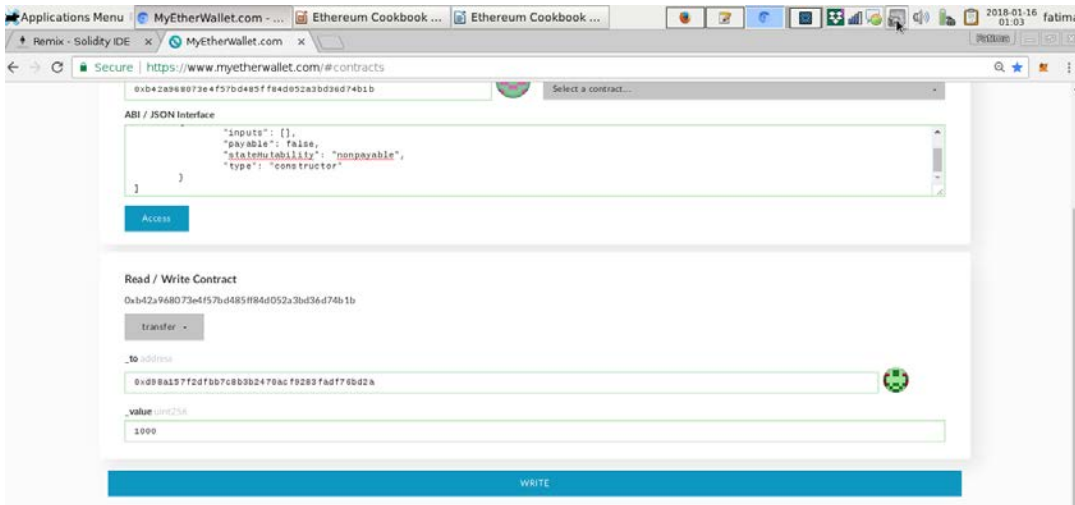


Figure 2.29: A screenshot displaying the code of the JSON interface that allows you to write

45. Click on **Generate Transaction**:

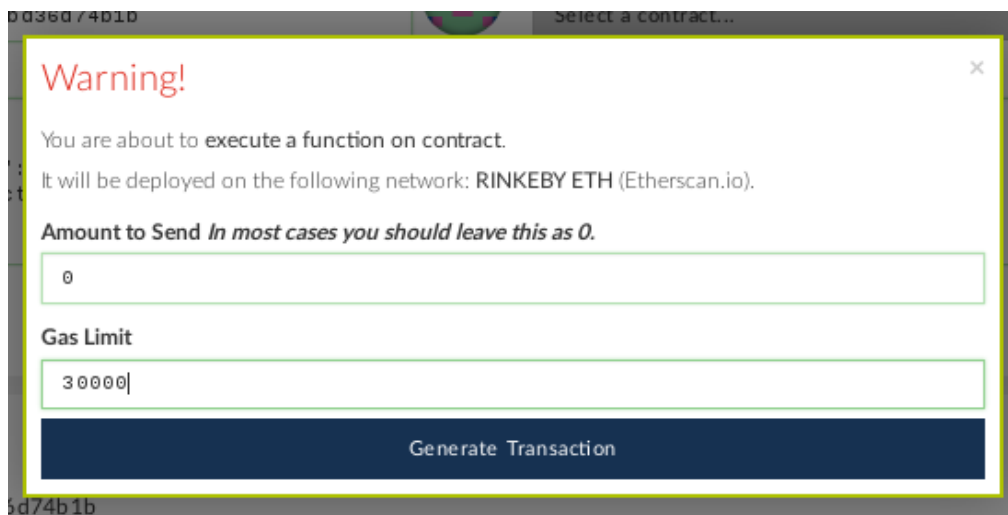


Figure 2.30: A screenshot that prompts you to generate transaction

46. Click on **Yes, I am sure! Make transaction:**

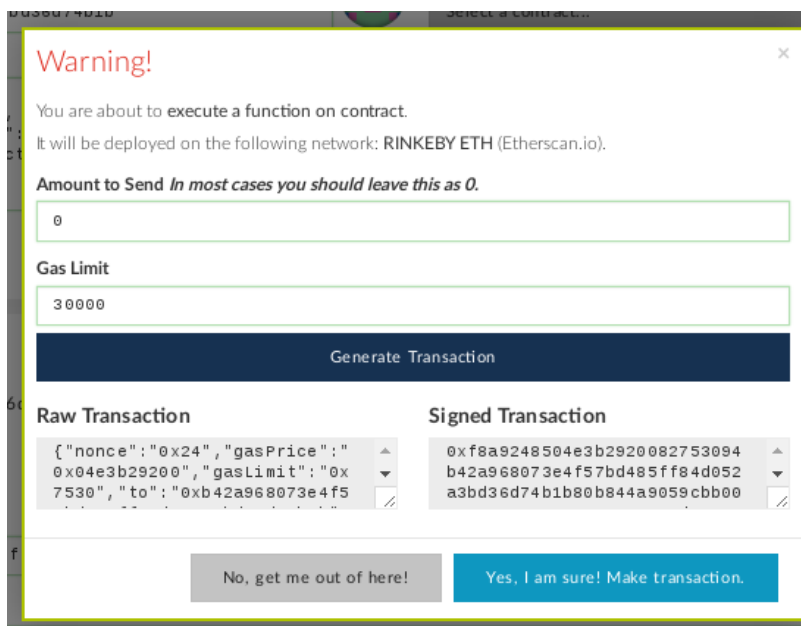


Figure 2.31: A screenshot that is asking for confirmation of the transaction

In this topic, we deployed and tested our first smart contract using MetaMask, Remix, and MyEtherWallet.

Summary

Now that you understand what smart contracts and tokens are, you should have a basic command of the Solidity language and an understanding of the deployment process, which allows you to write and deploy your own contracts. You have also learned about the difference between a traditional programming system and an Ethereum-based one, and about the different Ethereum blockchains.

Smart contracts are, in many use cases, enough by themselves. But for most cases, for example, an exchange, you need a frontend. That is where dApps take the scene. In the next lesson, we will explore the world of dApps.

3

Solidity Contracts

Learning Objectives

By the end of this lesson, you will be able to:

- Describe the basic framework of the Solidity language
- Use the Ethereum blockchain and the Ethereum network as a programming environment
- Write a smart contract in solidity
- Compile, deploy, and test smart contracts in the Rinkeby test network

In this lesson, you will write your first dApp and cover Oracle, Remix, MetaMask, Ganache, and web3.js.

Introduction

In the previous lesson, you learned about Solidity and smart contracts. We covered the basics of Solidity programming, including how to write, compile, deploy, and test a smart contract.

In this lesson, you will learn about dApps and Oracle. A dApp (decentralized application) is an application that runs on a decentralized network. An example would be a smart contract for an exchange that is running on the Ethereum platform with a web interface. dApps are important, because a web interface allows for easy interaction with the network.

By the end of this lesson, you will be familiar with the main technologies used to build dApps, and you will have gained some hands-on experience of building a dApp. You will be using the following technologies: Remix, MetaMask, Ganache, and web3.js.

In the Oracle section of this lesson, you will learn about the concept of an Oracle, and why Oracles are so important to the Ethereum ecosystem. You will learn how to work with a financial Oracle. A financial Oracle is an Oracle that specializes in financial data (for example, exchange rates). You will also learn about some new Solidity concepts, which will help you to understand how an Oracle works.

By the end of this lesson, you will be ready to integrate Oracle with your dApps.

Your First dApp

In this topic, you will build your first DApp: a simple voting contract. To do so, you will use MetaMask, Remix, Ganache, and web3.js. You should already know what MetaMask is. Remix is an IDE, Ganache is a blockchain simulator, and web3.js is a JavaScript library that is used to connect to the Ethereum network.

Once the DApp has been compiled and is running, MetaMask and Remix will no longer be required, because the DApp will connect directly to Ganache by using web3.js.

The following are the high-level steps to be followed to create a dApp:

1. Set up the development environment.
2. Write, compile, and deploy your smart contract.
3. Write a simple web page to interact with your contract.
4. Test your dApp.

Architecture of a dApp

For a dApp to connect to a blockchain, you require web3.js. web3.js is an API for Ethereum, written in JavaScript. It is an interface for the Ethereum JSON-RPC implementation. JSON-RPC is a remote procedural call protocol, encoded in JSON. Ethereum JSON-RPC is an implementation of JSON-RPC that is used for communication between an authenticated client and an Ethereum node. It is the main medium for applications to interact with the blockchain.

The architecture of a DApp is illustrated in the following diagram:

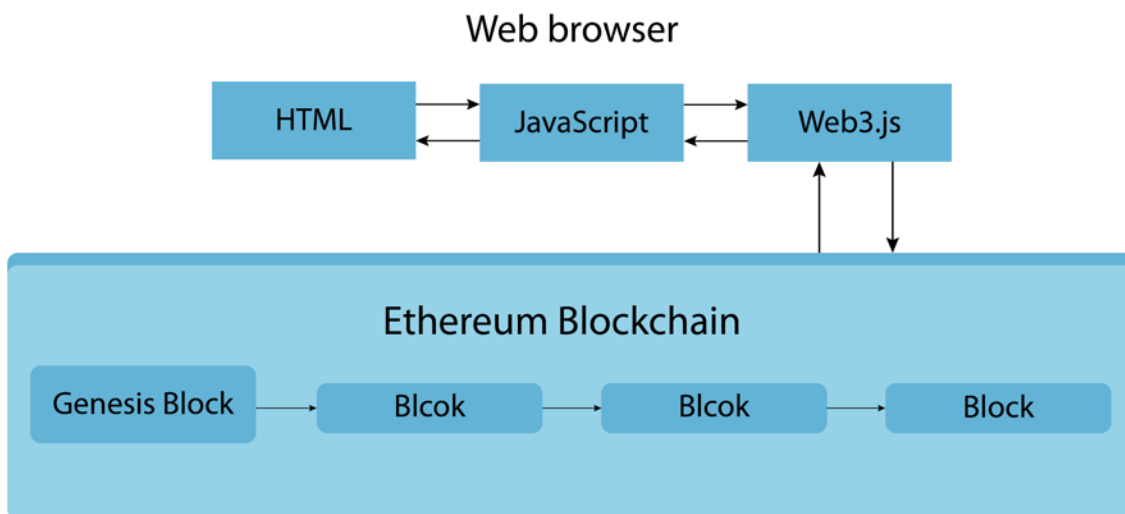


Figure 3.1: A diagram of the architecture of a dApp describing how the Ethereum blockchain interacts with the web browser through web3.js

A DApp with an HTML frontend uses the JavaScript web3.js library to connect to the Ethereum blockchain, via RPC. In our case, the blockchain will be our local Ganache client.

Ganache

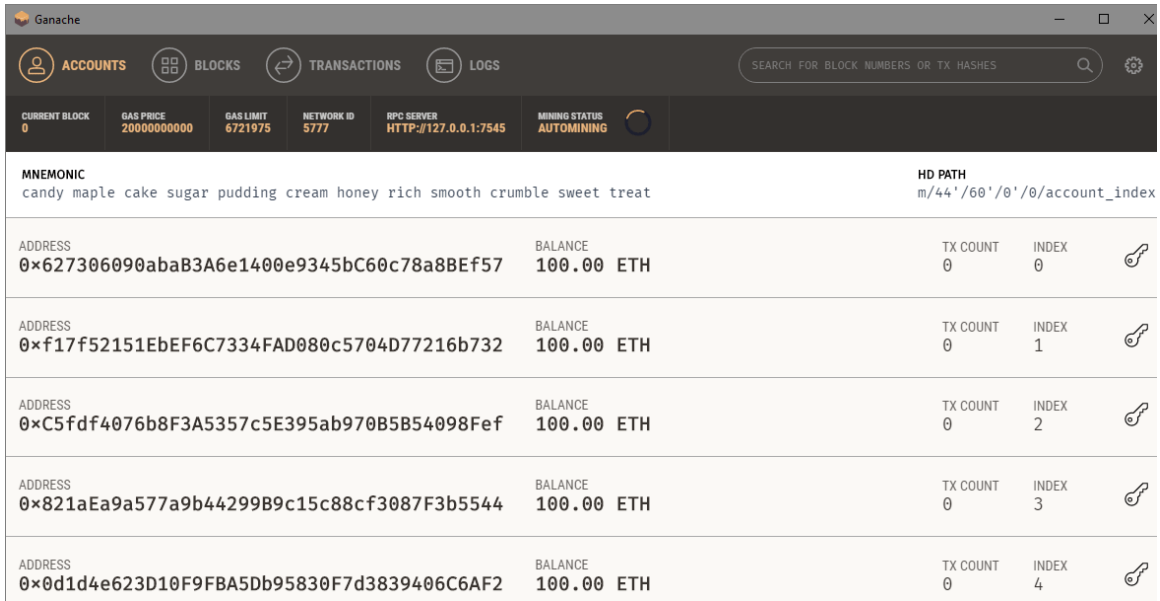
Ganache is a personal blockchain that is used for Ethereum development. You can use Ganache as a backend for dApps, and you can use it to deploy contracts and run tests. There are two versions of Ganache available:

- A graphical interface version (Ganache)
- A command-line version (Ganache CLI).

We will be using the version with the graphical interface. It contains four sections:

- Accounts (default)
- Blocks
- Transactions
- Logs

The **Accounts** section, by default, includes all of the addresses and their respective balances, as shown in the following screenshot:



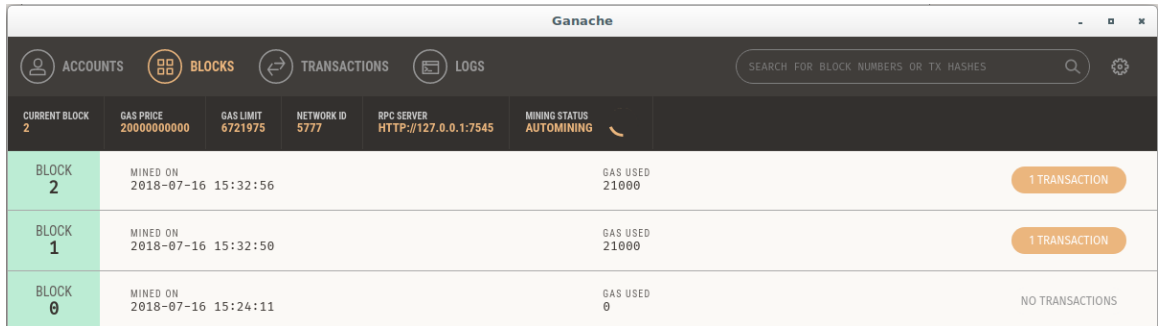
The screenshot shows the Ganache interface with the 'ACCOUNTS' section selected. The top navigation bar includes 'ACCOUNTS', 'BLOCKS', 'TRANSACTIONS', and 'LOGS'. Below the navigation bar, there are several status indicators: CURRENT BLOCK (0), GAS PRICE (2000000000), GAS LIMIT (6721975), NETWORK ID (5777), RPC SERVER (HTTP://127.0.0.1:7545), and MINING STATUS (AUTOMINING). The main content area displays the following information:

MNEMONIC: candy maple cake sugar pudding cream honey rich smooth crumble sweet treat
HD PATH: m/44'/60'/0'/0/account_index

ADDRESS	BALANCE	TX COUNT	INDEX
0x627306090abaB3A6e1400e9345bC60c78a8BEf57	100.00 ETH	0	0
0xf17f52151EbEF6C7334FAD080c5704D77216b732	100.00 ETH	0	1
0xC5fdf4076b8F3A5357c5E395ab970B5854098Fef	100.00 ETH	0	2
0x821aEa9a577a9b44299B9c15c88cf3087F3b5544	100.00 ETH	0	3
0x0d1d4e623D10F9FBA5Db95830F7d3839406C6AF2	100.00 ETH	0	4

Figure 3.2: A screenshot of the Accounts section, which is populated by default

The **Blocks** section lists all of the mined blocks, the gas that has been used, and the transactions, as shown in the following screenshot:

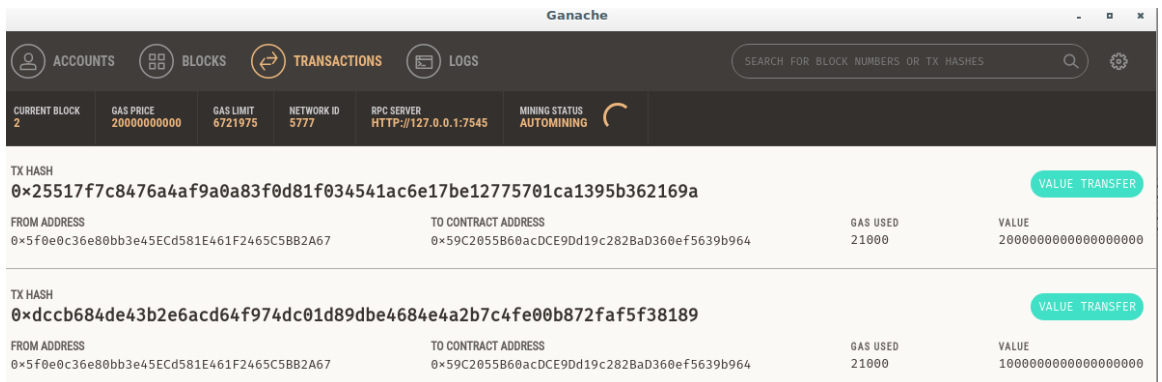


CURRENT BLOCK	GAS PRICE	GAS LIMIT	NETWORK ID	RPC SERVER	MINING STATUS
2	20000000000	6721975	5777	HTTP://127.0.0.1:7545	AUTOMINING

BLOCK	MINED ON	GAS USED	TRANSACTIONS
2	2018-07-16 15:32:56	21000	1 TRANSACTION
1	2018-07-16 15:32:50	21000	1 TRANSACTION
0	2018-07-16 15:24:11	0	NO TRANSACTIONS

Figure 3.3: A screenshot of the Blocks section that shows blocks 0, 1, and 2

All of the transactions are listed in the **Transactions** section, as follows:



TX HASH	VALUE
0x25517f7c8476a4af9a0a83f0d81f034541ac6e17be12775701ca1395b362169a	VALUE TRANSFER
FROM ADDRESS: 0x5f0e0c36e80bb3e45ECd581E461F2465C5BB2A67	TO CONTRACT ADDRESS: 0x59C2055B60acDCE9Dd19c282BaD360ef5639b964
	GAS USED: 21000
	VALUE: 2000000000000000000
TX HASH	VALUE
0xdccb684de43b2e6acd64f974dc01d89dbe4684e4a2b7c4fe00b872faf5f38189	VALUE TRANSFER
FROM ADDRESS: 0x5f0e0c36e80bb3e45ECd581E461F2465C5BB2A67	TO CONTRACT ADDRESS: 0x59C2055B60acDCE9Dd19c282BaD360ef5639b964
	GAS USED: 21000
	VALUE: 1000000000000000000

Figure 3.4: A screenshot of the Transactions section that shows two transactions

The **Logs** section displays the logs for all of the requests to the server:

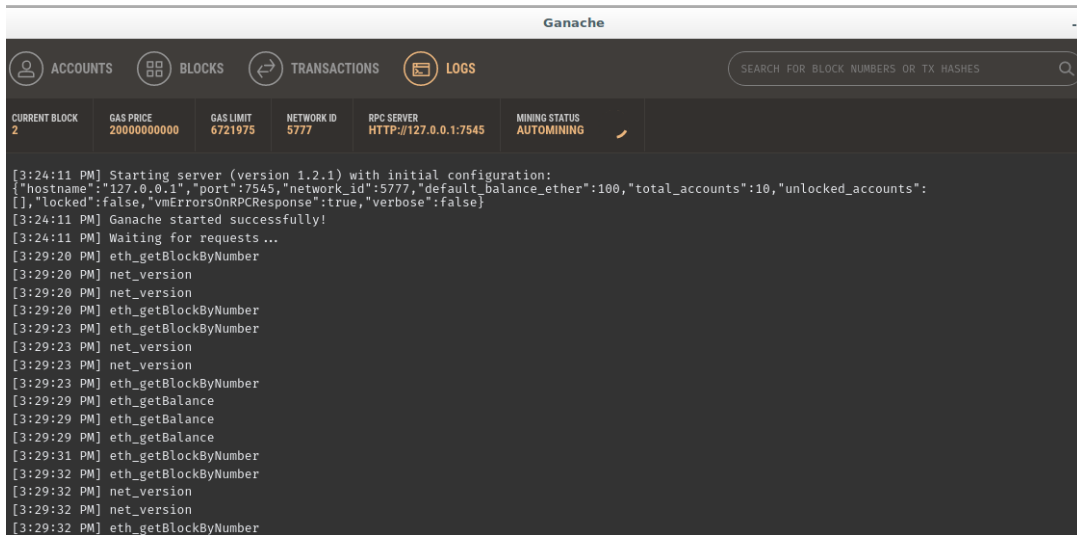


Figure 3.5: A screenshot of the Logs section that shows the request logs to the RPC server

Exercise 8: Using MetaMask to connect to Ganache

When running Ganache, by default, you start with 10 addresses, each with 100 ether, as shown in the following screenshot:

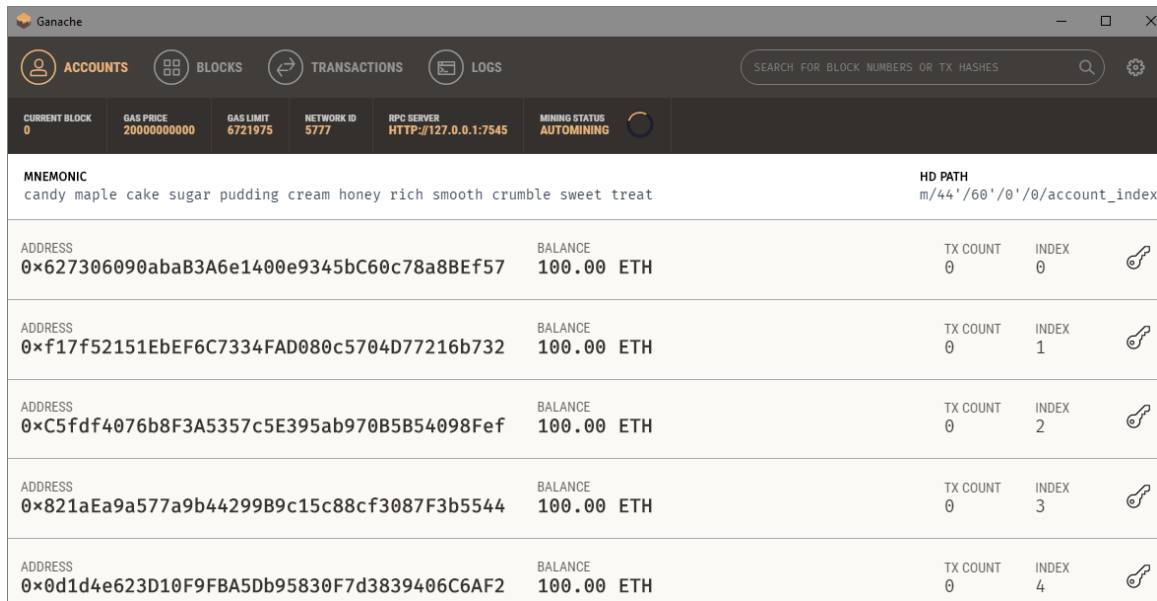


Figure 3.6: A screenshot of the default account section of Ganache

Since MetaMask is not aware of where the balance is, we have to import each address, one by one. So, you are tasked with configuring MetaMask to connect to Ganache and import the addresses. To do this, perform the following steps:

1. Configure MetaMask to use a private network (Custom RPC), as shown in the following screenshot:

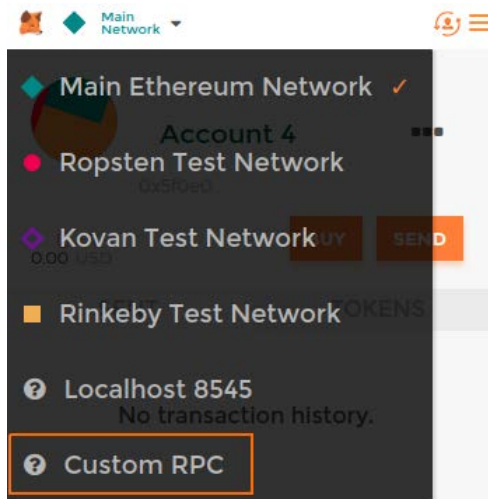


Figure 3.7: A dropdown that shows the different ways to configure MetaMask

You will be taken to the next screen, which prompts you to enter the New RPC URL. This should be available in Ganache:

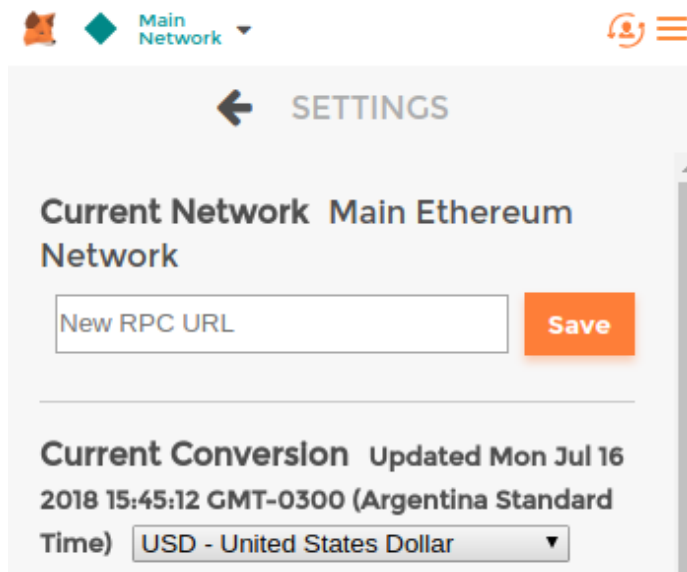


Figure 3.8: A screenshot of the screen that prompts you for a new RPC URL

2. Enter the New RPC URL and then click **Save**. You should now be connected to Ganache, and your balance should be zero, as indicated by the following screenshot:

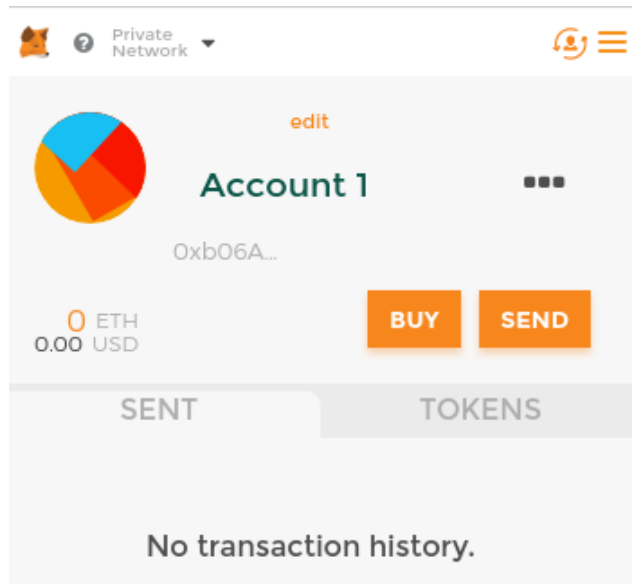


Figure 3.9: A screenshot that shows the transaction history of Account 1

3. Copy the private key of at least one of the addresses, and import it into MetaMask:
4. In the **Accounts** section, click on the key for the chosen address:

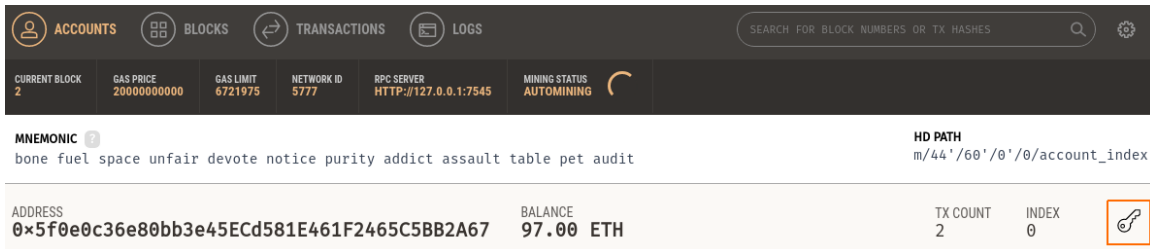


Figure 3.10: A screenshot of the Account section of Ganache that shows the balance as 97 ETH.

5. Copy the key and import it into MetaMask, as shown in the following screenshot:

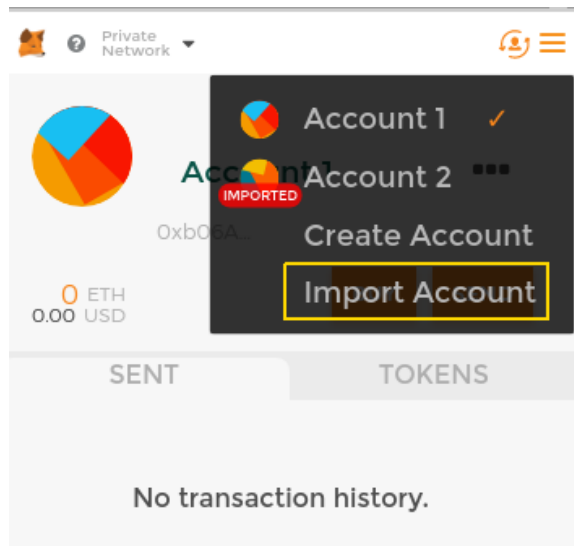


Figure 3.11: A screenshot that highlights the option to Import Account.

You should now have a balance in your account, as shown in the following screenshot:

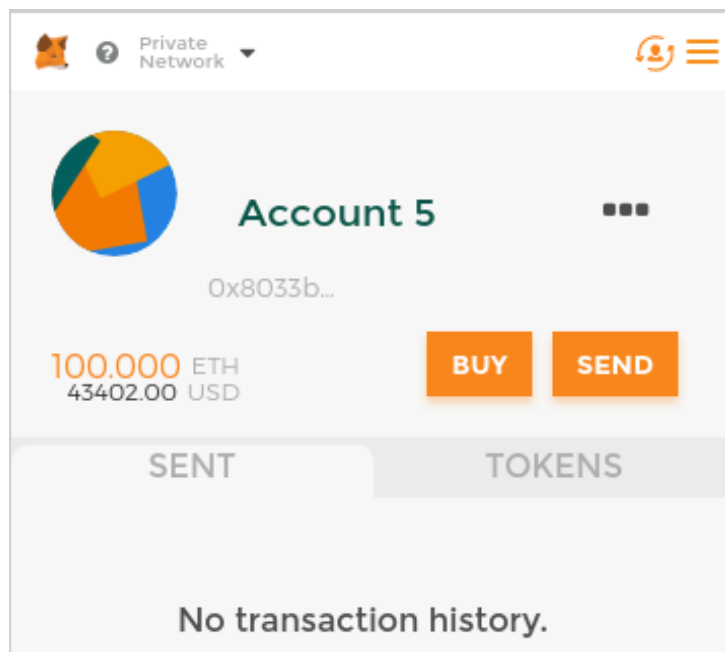


Figure 3.12: A screenshot of the balance in Account 5.

Voting Contract

In this subtopic, you will write a simple voting contract that will serve as the dApp's backend. When this contract is initialized, it will receive the following items:

- A list of valid candidates (addresses)
- A price (in Wei) to add a new candidate
- A price (in Wei) to vote for a given candidate

The contract will have six functions, as follows:

- **VotingContract**: A constructor function that initializes the candidates and prices to add/vote
- **AddCandidate**: Adds a new candidate (the candidate must not exist, and the price must be correct)
- **VoteForCandidate**: Votes for a valid candidate (the candidate must exist, and the price must be correct)
- **CandidateExists**: Returns whether a candidate exists
- **HasVoted**: Returns whether an address has voted
- **VotesForCandidate**: Returns the number of votes cast for a given candidate

The contract will have six variables, as follows:

- **uint PriceToAdd**
- **uint PriceToVote**
- **address[] Voters**
- **mapping (address => uint) public votes**
- **address[] public Candidates**
- **uint public numberOfCandidates**

The code for the contract is as follows:

```
pragma solidity ^0.4.18;
contract VotingContract {
    uint PriceToAdd;
    uint PriceToVote;
    address[] Voters;
    mapping (address => uint) public votes;
```



```
//[...]
function VotesForCandidate(address Candidate) view public returns (uint) {
    return votes[Candidate];
}
}
```

Compiling and Deploying Contracts

Using Remix, compile and deploy your contract. Call the contract Voting.sol. Upon deploying your contract, you will have to provide a list of addresses for the candidates, in the following format:

```
["address1", "address2", "address3", "addressN"]
```

The address can be any valid address.

You also have to specify the price to vote and the price to add a new candidate (in Wei).

The result should be something like the following:

```
["address1", "address2", "address3", "addressN"], 100, 100
```

When your contract has been compiled, copy the contract's address. You will need this address so that your DApp can connect to the contract:

Deployed Contracts

VotingContract at 0x388...e4320 (blockchain)

AddCandidate	address Candidate
VoteForCandidate	address Candidate
CandidateExists	address Candidate
Candidates	uint256
HasVoted	address Voter
numberOfCandidates	
votes	address
VotesForCandidate	address Candidate

Figure 3.13: A simple contract

A Simple Web Page

Now that Ganache is running and a smart contract has been compiled, you will build a small site to connect to the DApp and cast votes. The site will include an HTML side and a JavaScript side. You will begin by editing the JavaScript side, according to your contract's data. **YourContractAddress** is the contract address that you just copied.

You should name the file **index.js**.

If you look closely at the code, you will notice that it is built on top of web3.js.

Now, name the HTML file **index.html**; it does not require any changes.

The HTML code is as follows:

```
<!DOCTYPE html>
<html>
<head>
  <title>My First DApp</title>
  <link href='https://fonts.googleapis.com/css?family=Open+Sans:400,700'
rel='stylesheet' type='text/css'>
  <link href='https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.
min.css' rel='stylesheet' type='text/css'>
</head>
<body class="container">
  <h1>My Voting Contract</h1>
  <div class="table-responsive">
    <table class="table table-bordered">
      <thead>
        <tr>
          <th>Address</th>
          <th>Votes</th>
        </tr>
      </thead>
      <tbody id="tbody">
      </tbody>
    </table>
```

```

</div>
<input type="text" id="candidate" />
<a href="#" onclick="voteForCandidate()" class="btn btn-primary">Vote</a>
<br><br>
  <input type="text" id="addCandidate" />
  <a href="#" onclick="addCandidate()" class="btn btn-primary">Add
Candidate</a>
<br>
</body>
<!-- <script src="bignumber.js"></script> -->
<script src="https://cdn.rawgit.com/ethereum/web3.js/develop/dist/web3.
js"></script>
<script src="https://code.jquery.com/jquery-3.1.1.slim.min.js"></script>
<script src="./index.js"></script>
</html>

```

Now, open **index.html**; you should see something like the following screenshot:

My Voting Contract

Address	Votes
0x6eb9cf6e73febe0bef85f02127d86a2a183cf6ea	1
0xaf25c62132b64dad49f0014b8921e50334c7cec7	0
0x607a14e0c86b33fa96dbd834e020959fa3eb005f	0
0x9b7e1f9b3abcefb5b5412a60a13eaf3bd80bba69	0

Figure 3.14: A screenshot of the Voting application.

You now have a running dApp, and you can test it.

A Simple Hello World Voting Application

Candidate	Votes
0x89Da55E3d49dbDF28bcB905EF4f813366D5b0D20	0
0x76871a0aA755fc8Ed124469CbbA8e565FAee484	0
0x85d4aBA930041dc905aeAbd3182937A4530a45B	0

Figure 3.15: A screenshot of the Voting application while casting a vote.

In this topic, you built your first dApp. You used MetaMask, Ganache, Remix, and web3.js. You should now understand the relationships between the different tools. In the next topic, you will learn about Oracle and how to interact with them.

Using an Oracle

An Oracle is a third party that you communicate with when you need outside-world data. For example, when you need the current rate for ETH-BTC, you can ask an Oracle. The Oracle will answer by sending you the requested information, and will charge you a small fee. We will be using the Ethernity Financial Oracle in this topic.

Oracles are the only medium through which a dApp can get information from outside of the blockchain. The following diagram illustrates how an Oracle works:

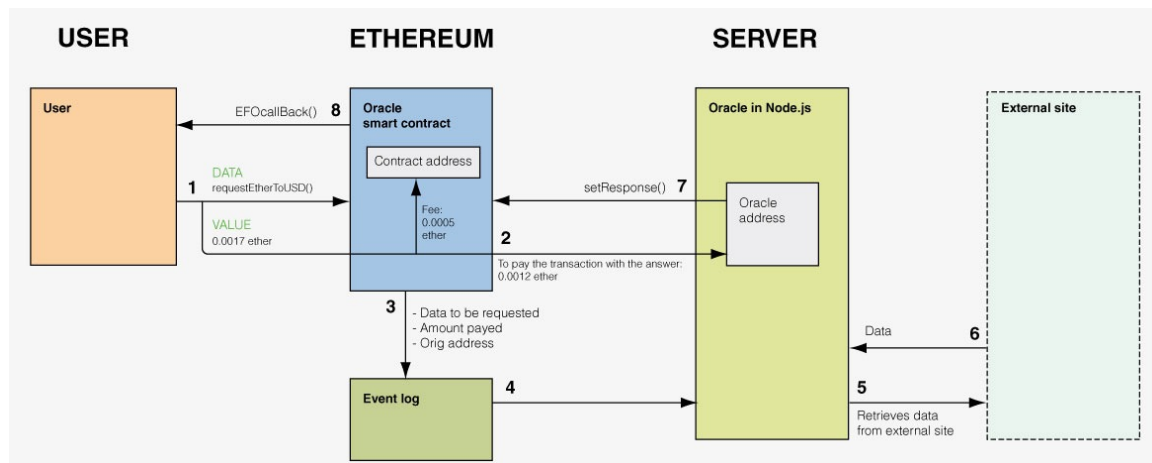


Figure 3.16: A diagrammatic representation of how dApps interact with Oracle

Interface

In order to use the Financial Oracle from within a contract, you must copy, or import, the following interface into your contract:

```
contract EthernityFinancialOracle {
    event Request (string _coin , string _againstCoin , address _address ,
    uint _gasPrice , uint _gasLimit);
    // Requests (you only need to have the ones that you will use it)
    function requestEtherToUSD(bool _callback , uint _gasPrice, uint _
    gasLimit) payable;
    function requestCoinToUSD(string _coin , bool _callback , uint _gasPrice
```

```

, uint _gasLimit) payable;
    function requestRate(string _coin, string _againstCoin , bool _callBack ,
uint _gasPrice , uint _gasLimit) payable;
    // Following are optionals. You can have the ones that you will use
    function getRefund();
    // Getters
    function getResponse() public constant returns(string _response);
    function getPrice(uint _gasPrice , uint _gasLimit) public constant
returns(uint _price);
    function getBalance() public constant returns(uint _balance);
    uint public feePrice;
    uint public gasLimit;
    uint public gasPrice;
}

```

Usage

The basic usage of the Rinkeby test network is as follows:

```

function callOracle {
    // Define Oracle (using Rinkeby address) invoking the interface
    EthernityFinancialOracle EFOracle =
EthernityFinancialOracle(0x7e106c6e896ea801824da24386d7d59311235ec7);
    // Make request
    EFOracle.requestEtherToUSD(true);
}

// Function to be called by EFOracle when request is ready
function EFOcallback(string _response) {
    require(msg.sender == 0x7e106c6e896ea801824da24386d7d59311235ec7);
    // Here you can process the received _response
}

```

Payment

It is mandatory to pay for the Financial Oracle in order to trigger a transaction that provides the response to your query. The requested price includes a fixed fee for the Financial Oracle, plus an amount that the Financial Oracle will use to pay for the gas for the transaction.

There are two ways to pay for your requests, as follows:

- At any moment prior to making your request, you can deposit Ether into the contract a by simply clicking **Send**. The Ether will be automatically stored as a credit balance for your address. You can make as many additional requests as you desire from the same address, as long as you have a large enough balance. You can check your balance at any time using the `getBalance()`; command. Any exceeding balance that has not been used to pay for your requests can be refunded at any time using the `getRefund()`; command.
- You can send a payment with each of your requests. This can be done from Solidity, as follows:

```
requestEtherToUSD.value(_payment)(true);
```

In the preceding command, `_payment` should indicate the amount to send (in Wei).

You can also create a manual transaction from [Myetherwallet](#), or any other system that allows you to send a value.

Calculating Payments

The price of each request is a single value, composed of three variables: `feePrice`, `gasPrice`, and `gasLimit`.

The first variable is the amount that the Financial Oracle will receive as payment, and it cannot be modified. It is currently set to 0.0005 ethers, but that may change in the future. The second and third variables are the values that will be used by the Financial Oracle as network gas fees, used to send the answer back to you. You can choose your own values, or you can rely on the default values (40 Wei for the price and 50,000 for the limit = 0.002 ether). Note that the Financial Oracle will retain any remaining gas that is left over from the transaction. The total price, using the default values, is 0.0025 ether.

There are two ways to calculate the amount to pay, depending on whether you use the default values for the **gasPrice** and **gasLimit**:

- Calculate the requested price by using the default values for the **gasPrice** and **gasLimit**, as follows:

```
getPrice();
```

This will tell you how much money will be taken from your balance with each default request (or, how much you have to send with each request).

- Calculate the requested price for a specific **gasPrice** and **gasLimit**:

```
getPrice(gasPrice,gasLimit);
```

This will provide you with the total price for the gas, plus the price of the fee, which will be the total amount deducted from your balance (or sent with the request) when you make a request with the specific values. To make a request that specifies the gas price and gas limit, use the following function:

```
requestEtherToUSD(true , gasPrice , gasLimit);
```

This will tell Financial Oracle to use these values to pay the fee for the transaction with your required answer. In order for this to work, you must send the correct amount to Financial Oracle; you can send it with your request, with a command like the following:

```
requestEtherToUSD.value( getPrice(gasPrice,gasLimit) )(true , gasPrice , gasLimit);
```

Note that none of the request prices will be refunded, even if the request was not successful or the answer did not consume all of the gas. Any value that surpasses the request price will be stored as a credit for the address, and can be used or refunded at any time.

Request Types

There are two ways to get a request: in a passive way, or in an active way. You can choose the kind of request that you'd like to make by using the first argument of the request; use **true** for a passive callback, and **false** (or just nothing) for an active one.

In the passive request type, the Financial Oracle will send you the answer by calling **EF0callback(string _response)** in your contract, with the answer included in the **_response** variable.

The advantages of this method is that it is private, and you can regulate the cost of the call by making your callback function fit your own needs. A disadvantage is that you have to use a contract to call the Oracle.

In Active Request type, the Financial Oracle will store the answer in an internal mapping, and it will write the event response (**address _address, string _response**) to the blockchain. You should watch for the event, and then retrieve the data from the event (or, by calling **getResponse()**).

Advantages of this method include that it can be cheaper than using a callback function (approximately 30,000 units as consumed/40.000 the first time) and you can make calls from a simple address (no need to be a contract). Some disadvantages include that the data will be public, and you will have to watch for the event before reading the data.

- To get the Ether price in USD, use the following function:

```
requestEtherToUSD (bool _callBack, uint _gasPrice , uint _gasLimit);
```

- To retrieve the rate of any coin in USD, use the following function:

```
requestCoinToUSD (string _coin , bool _callBack, uint _gasPrice , uint _gasLimit);
```

- To retrieve the rate of any coin against any other coin, use the following function:

```
requestRate (string _coin , string _againstCoin , bool _callBack, uint _gasPrice , uint _gasLimit);
```

Note that if you send a new request before receiving the answer to the first one, the second one will overwrite the first one, but you will be charged for both of the requests.

Functions and Getters

This will create a request for the actual price of the Ether in USD. All of the parameters are optional:

```
requestEtherToUSD (bool _callBack, uint _gasPrice , uint _gasLimit)
```

If **_callBack** is true, the answer will be a callback. If it's false (or absent), the answer will be stored in a mapping, and also in a log event.

If the **_gasPrice** and **_gasLimit** are specified, they will be used for the Financial Oracle to make the callback (or store the answer). If they are not specified, the Financial Oracle will use the default values. Note that you must send the total value (for the gas, plus the fee) with the request or fill your balance by sending ether to the contract. You can also consult how much you have to pay in any case with the corresponding getters, described as follows:

```
requestCoinToUSD (string _coin , bool _callBack , uint _gasPrice , uint _gasLimit)
```


To request the rate of any Cryptocurrency in USD, you have to specify the coin in the first parameter. The following parameters are optional, just like in the previous case:

```
requestRate (string _coin , string _againstCoin , bool _callback , uint _
gasPrice , uint _gasLimit)
```

To request the rate of any coin against any other coin, you must specify both coins in the first two parameters.

- The **getRefund()** command will send back your available balance.
- **getPrice()**: This shows the total price that you must pay for each default request. You can send the value with a request, or you can make sure that it is stored in your balance in the Oracle by sending ether to it.
- **getPrice(gasPrice, gasLimit)**: This shows the total price of each request if you specify a gas limit and gas price for the callback. You can choose both values, ensuring that they are in account that they will be used to call to your callback function or to write a mapping and a log event. If the amount is not high enough for the call, the transaction with the answer will fail, and you will lose your payment.
- **feePrice()**: This shows the actual fee that will be charged with each request. It is the amount that you must pay for the service of the Financial Oracle, and it is a part of the total price to pay for each request (the other part is the gas that is required).
- **gasPrice(), gasLimit()**: This shows the default gasPrice and gasLimit that will be used to send you the result (or to store the result in a mapping and a log), except that you specify the price and limit you want. This is only a part of the total price of the request (the other part is the fee).
- **getBalance()**: This shows your available credit, which can be used for requests
- **getResponse()**: This shows the answer to the last request from your address (it only works when you specify the **_callback** as false, so the answer is logged and stored in a mapping)

Consider the following example contract. The contract name is Caller. A simple way to use this example contract is detailed as follows:

- **getPrice()** will show the value to send with the request.
- **request()** will generate a request (with the request, you must pay the value that you calculated with **getPrice()**).
- **response()** will show the response from the Oracle

An advanced way to use this example contract will be detailed as follows. You can choose any values that you consider necessary for **gasPrice** and **gasLimit**:

- **getPrice(gasPrice, gasLimit)** will show the total price that you must pay for each request
- **request(gasPrice, gasLimit)** will generate a request (with the request, you must pay the value that you calculated with the preceding function)
- **response()** will show the answer from the Oracle

You can check the process at the Oracle contract.

The Caller code is as follows:

```
pragma solidity ^0.4.18;

contract EthernityFinancialOracle {
    function requestEtherToUSD(bool _callBack , uint _gasPrice, uint _
gasLimit) payable;
    function getPrice(uint _gasPrice, uint _gasLimit) public constant
returns(uint _price);
    event Request (string _coin , string _againstCoin , address _address ,
uint _gasPrice , uint _gasLimit);
}

contract Caller {

    string public response; // Public getter to see the answer
    address public oracleAdd; // Oracle address
    address public owner;

    modifier onlyOwner{
        require(msg.sender == owner);
        -;
    }

    function Caller() {
        owner = msg.sender;
        oracleAdd = 0x7e106c6e896ea801824da24386d7d59311235ec7; // Rinkeby
```

```

address
    }

    function EF0callback(string _response) {
        require(msg.sender == oracleAdd);
        response = _response;
    }

//[...]
}

```

Every request is made up of two main transactions.

The first transaction is the request, which originates from the user (or the user's contract) and is sent to the Oracle address:

Fee: The fee for the Ethereum network is the same as that of any Ethereum transaction. Some wallets calculate this automatically. If the calculation is done manually, it's recommended to enforce a gas limit of 120,000 units (between 75,000 and 105,000 units will be used). The gas that is not used will be refunded to the originating address, just like in any other Ethereum transaction.

Data: The transaction data is the call to the request function in the Oracle (`requestEtherToUSD`, `requestCoinToUSD`, and so on). The function can be called with or without arguments. The first argument (`_callback`) is a Boolean that specifies the desired type of response. If it's true, the Oracle will try to call an `EF0callback(string)` function to the originating address, with the answer. If it's false or is not set, the Oracle will store the answer in its own address and generate an event log.

The second and third arguments define the price that will be charged for the request. This price is composed of a fee, plus an amount that will be used to send the answer back. If they are not present, Oracle will charge the default price (which can be retrieved with `getPrice()`), and it will use the default values of `gasPrice` and `gasLimit` to send the answer (which can be retrieved with the `gasPrice()` and `gasLimit()` getters). If they are set, Oracle will use the specified values to set the price for its answer. The total price that will be charged of the originating address can be retrieved with `getPrice(_gasPrice, _gasLimit)`, which calculates the price to pay to the Oracle based on those values, plus the fee for the Oracle.

A delicate point to note is that if you pay more than is required, the excess will be added to your account balance. The Oracle will only charge you the gas limit and gas price that you have specified (or the default amounts, if you didn't specify custom amounts), plus the fee for the Oracle. If the Oracle's answer consumes less gas than you specified, the excess will not be refunded, nor will it be stored as a balance; it will be returned to the Oracle by the Ethereum network, and will be used for administrative purposes.

Value: If you already have a balance in your Oracle account and it's enough to pay for the transaction, you don't have to send anything with the request. On the contrary case, the request should be accompanied of a value to pay the Oracle. The price of the request will depend on the arguments that you have passed to the request function, as explained previously. The Oracle will take a fee as payment, and it will use the specified gas price and gas limit to send the answer to you. As we noted previously, if you send more than the calculated price, the excess will be stored as a part of your balance. If the answer consumes less gas than specified, the excess will not be refunded to the user, but will remain in the Oracle for administrative purposes.

The second transaction is the answer:

Once the Oracle has processed your request, it will retrieve the answer and send it in an Ethereum transaction. This transaction originates in a third address (registered as `oracleAddress`), which will send the answer to the Oracle contract.

Fee: To set the gas price and gas limit required for the Ethereum network to send the answer, the Oracle will use either its default values or the values that the user passed with the request. If the gas price or limit is set too low, the transaction can fail.

Data: The `oracleAddress` will call the `setResponse` function to the Oracle contract, with the data of the response in the `_response` string.

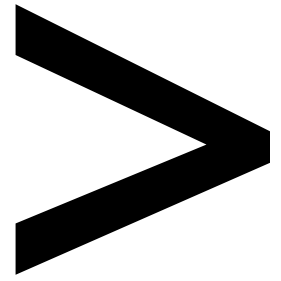
If the Boolean argument sent with the request is false or absent, the Oracle will generate the event `Response(address, string)` with the originating address and the answer as the values, and will store the value in a mapping that can be accessed with the getter `getResponse()`, called from the originating address. This transaction will generally consume 40,000 units of gas the first time it is used from an address, and 30,000 units of gas for the following transactions.

If the first argument that is sent with the request is true, the Oracle will make an internal call to the `ECOCallBack(string)` function in the originating address of the user contract. In that case, the gas consumption of this second transaction, including the internal one, will depend upon the function in the originating address. The gas cost can be calculated by creating trials in Kovan or Rinkeby.

Summary

In this lesson, you learned about dApps, and then you built one. You also learned about Ganache and web3.js. You used Remix and MetaMask to connect to your own private blockchain. We then covered Oracles and how to use them. You should now understand their functionality, and you should understand how to make calls in them. You should also know the differences between functions and getters, and be able to differentiate between the types of requests.

Now, you are ready to continue on your Ethereum journey on your own.



Index

About

All major keywords used in this book are captured alphabetically in this section. Each one is accompanied by the page number of where they appear.

A

accessed: 3, 80
accordance: 3
according: 3, 70
account: 1, 9-14, 16, 19,
64, 66-67, 77, 80
achieved: 17
achieves: 4
actions: 20
active: 75-76
advanced: 78
algorithm: 5, 7-9
altcoin: 34
amount: 27, 43-44, 49,
74-75, 77, 79-80
asymmetric: 5, 8-9, 28
athenian: 20

B

backend: 61, 68
barcode: 16
barcodes: 15
bignumber: 71
bitcoin: 2, 4, 9, 11-12
bitcoinqt: 12
blockchain: 1-5, 9-11, 16,
18-21, 23-24, 26, 28,
31-35, 41-42, 45, 47,
49, 59-61, 72, 76, 81
blocks: 2, 4, 19-20,
24, 26, 34, 62-63
byzantine: 16-17

C

caesar: 7
checksum: 14
chrome: 47-48
chromium: 47
cipher: 5, 7

classify: 18-19
closely: 70
coding: 32
collection: 42-45
compile: 31, 39-40,
59-60, 69
configure: 65
consensus: 2-3
context: 35
contexts: 2
contract: 4-5, 10, 20, 22,
31-38, 41-42, 44-47,
52, 54, 57, 59-60,
68-70, 72, 74-80
costly: 3
coupons: 37
course: 4, 12-13
covered: 4, 60, 81
cracking: 8
create: 4, 12-13, 20,
27-28, 34-35, 37, 40,
44, 46, 60, 74, 76
credit: 50, 74-75, 77
critical: 3
cryptocoin: 77
cryptology: 1-2, 5,
7, 9, 28, 32
currency: 11, 36
cypher: 7

D

debugging: 39
decide: 17, 50
declare: 43
decode: 6-8
decrypt: 3, 5, 8-9
define: 41-42, 73, 79
deletes: 3
delicate: 80
deliver: 17
depend: 80

deploy: 10, 31, 45-47,
57, 59-61, 69
deposit: 74
digital: 17
diverge: 3
dropdown: 12, 49, 65
during: 36

E

ecosystem: 2, 60
eforacle: 73
electrical: 6
electronic: 4
electrum: 12
encoded: 6, 61
encrypt: 3, 5, 8-9
encryption: 7-9
enigma: 8
eth-btc: 72
etherchain: 26
ethereum: 1-2, 4-5, 9-11,
13, 15-16, 18-21, 26-28,
31-35, 37, 40-41, 45-47,
49, 57, 59-61, 71, 79-81
ethernity: 72
ethers: 74
etherscan: 12, 21-23, 26
ethminer: 19
ethplorer: 26
exchange: 57, 60
exodus: 19
expensive: 10
experience: 60
explained: 13, 39, 80
explore: 33, 57
explorer: 26, 28, 34
explorers: 26, 35
extends: 19
extension: 47-48
extremely: 11

F

facebook: 15
faucet: 14
features: 11-12, 32, 39
feepriice: 73-74, 77
framework: 31, 59
frontend: 57, 61
function: 3, 19-20, 34,
37-38, 42, 44, 54,
68-69, 72-73, 75-80
functions: 4, 11-12, 34,
36, 41, 68, 76, 81
funder: 43
funders: 43
further: 17
future: 74

G

gaslimit: 72-79
gasprice: 72-79
gastracker: 26
generating: 27
genesis: 24, 34
german: 8
gesture: 6
getbalance: 73-74, 77
getprice: 73, 75, 77-79
getrefund: 73-74, 77
global: 41-42, 45
goleft: 43
google: 15
googleapis: 70
goright: 43
gostraight: 43

H

hackable: 11
hardware: 8, 11
hashes: 20-21, 24, 26

I

identical: 3-5
impact: 2, 23
implement: 8, 37
import: 65-67, 72
important: 17, 28,
36, 44, 60
include: 20, 44, 70, 76
included: 4, 19, 26, 75
indicate: 74
insecure: 9
insert: 37
inside: 4, 14, 27-28, 41
instance: 17
integer: 41, 45
interface: 54, 56,
60-62, 72-73
internal: 76, 80
internet: 16, 27
introduced: 5
investment: 2
invoking: 73

J

javascript: 32, 40,
60-61, 70
jquery: 71

L

language: 28, 31-32,
40, 45, 57, 59
ledger: 2-3, 24
libraries: 32
library: 60-61
linear: 24
litecoin: 4

M

machine: 18-19
mainnet: 45
maintain: 3
majority: 17
managing: 43
mandatory: 74
manual: 74
manually: 79
mapping: 17, 36-38,
43-45, 68, 76-77, 80
meaning: 6
measure: 36
medium: 61, 72
memory: 41
mentioned: 10
message: 5-8
messages: 6, 8, 51
messengers: 17
metamask: 19, 45,
47-51, 55, 57, 59-60,
64-67, 72, 81
method: 6, 8, 75-76
mileage: 37
military: 9
million: 9
miners: 19, 26
mining: 1, 4, 9, 16,
19-20, 24, 32, 34
minutes: 4
mistakes: 8
mobile: 4, 11
models: 16
modern: 28, 42-43
modify: 2-3, 20,
33, 44, 74, 78
moment: 74
monetary: 2
multiple: 10
multisign: 10
myriad: 28

mytoken: 37-39, 44

N

nakamoto: 12

nature: 33

necessary: 78

needed: 2, 6, 8-10, 12-13

network: 1, 3-5, 9-12, 14,
16, 19-21, 26-28, 31-33,
35, 37, 45-47, 59-60,
65, 73-74, 79-80

non-faulty: 17

nonsense: 7

notice: 70

number: 2-5, 8, 19,
33, 37-38, 44, 68

numerals: 6

O

objectives: 1, 31, 59

observer: 7

offline: 11

onclick: 71

online: 11-12

operation: 19, 35

operations: 10, 34-36

option: 67

optional: 76-77

optionals: 73

options: 55

oracle: 59-60, 72-80

oracles: 60, 72, 81

P

paired: 5

paradigm: 28, 33

parameter: 77

parity: 18-19

partially: 14

particular: 3

parties: 3, 6

passed: 20, 80

passive: 75

password: 12-13, 27, 49, 56

pasted: 15

pattern: 26

payable: 72-73, 78

payment: 10-11,
45, 74, 77, 80

payments: 2, 4, 10, 74

people: 4-5, 12, 33

perform: 1, 10, 16, 19-20,
35-37, 39, 44, 47, 65

performed: 3

performs: 19

perhaps: 11

permanent: 10

permitted: 3

personal: 61

physical: 18, 24

physically: 17

pieces: 11

pioneers: 11

pivotal: 32

platform: 60

points: 4, 34

popular: 12, 32

populated: 62

pop-up: 48

portion: 17

possesses: 5

possible: 19

potential: 2

powerful: 4

practical: 1, 4, 16, 33

practice: 7, 33

pragma: 68, 78

preceding: 40, 49, 74, 78

prefer: 17

presence: 17

present: 79

preset: 3

previous: 2, 19-21, 24,
32, 45, 60, 77

previously: 13, 34, 39, 80

prices: 68, 75

pricetoadd: 68

printed: 11

private: 3-5, 9-11, 13, 16,
27, 35, 65-66, 75, 81

probably: 26

problem: 3, 10, 16-17

problems: 4

procedural: 8, 61

process: 4, 7, 9, 19,
45, 57, 73, 78

processed: 4, 24, 80

processes: 46

processing: 10, 33, 41

processor: 10

productive: 45

program: 10, 31, 33, 35

progress: 15

prompts: 56, 65

proposal: 3

proposed: 3

protocol: 61

provide: 15, 69, 75

provides: 25, 74

providing: 25

public: 2-3, 5, 9, 11,
13, 15-16, 20, 27-28,
35, 37-38, 44,
68-69, 73, 76, 78

publicly: 6

publish: 15

purporting: 3

purpose: 4

purposes: 80

putting: 33

python: 32, 40

Q

quantity: 56
question: 15

R

radically: 19
random: 56
rapidly: 3
rather: 3, 28
rawgit: 71
reaches: 27
reader: 16
reading: 20, 34-35, 76
reality: 4
rearranges: 7
reasons: 27
receive: 4, 10-11, 14, 16, 19,
26, 28, 34-35, 68, 74
received: 2, 4, 17, 24, 73
receiver: 27
receives: 10, 28
receiving: 4, 21, 26-28, 76
recently: 32
recipient: 8, 27
recipients: 7
record: 10, 35
recorded: 20,
23-24, 26, 35
recording: 35
records: 3
refers: 2
refunded: 74-75, 79-80
registered: 80
regulate: 75
regulated: 4
reject: 2
related: 35
relevant: 3
remain: 80
remained: 9

remaining: 74
remains: 42
remember: 28
remind: 32, 40
remote: 61
replaced: 6
replicate: 8
request: 15, 64, 72-80
requested: 72, 74-75
requests: 15, 64, 72,
74, 76-77, 81
require: 3, 20, 27, 37-39,
61, 70, 73, 78-79
required: 14, 60, 75, 77, 80
requires: 3, 10
research: 9
respective: 62
response: 15, 73-80
result: 2, 69, 77
retain: 74
retreat: 17
retreating: 17
retrieve: 42, 76, 80
retrieved: 79
return: 42, 69
returned: 42, 80
returns: 19, 42,
68-69, 73, 78
reveal: 7
reveals: 3
review: 38
reward: 19
richer: 12
right-hand: 52
rights: 4
rigorous: 3
rikeby: 16
rinkeby: 10, 12, 14, 19, 26,
31, 45, 59, 73, 78, 80
ripple: 4
robots: 34
roughly: 19

rounds: 10
running: 10, 16, 18-19,
32, 60, 64, 70-71

S

satoshi: 12
scanner: 16
scopes: 40, 45
screen: 13, 51, 53, 65
screenshot: 13-15,
21, 25, 43, 48-49,
51-57, 62-67, 71
script: 71
scroll: 48, 53
search: 26, 28
second: 44, 74, 76, 79-80
secret: 5-7, 9
section: 53, 60, 62-64, 66
sections: 36, 44, 62
secure: 3, 9, 11
security: 5, 11, 27
select: 12, 20, 49, 54, 56
sender: 8, 10, 28, 38-39,
44, 73, 78-79
senders: 7
sending: 4, 21, 23,
26-27, 72, 76-77
separated: 17
sequence: 10, 21, 24
sequences: 5
server: 64
service: 18, 28, 77
setting: 1
shared: 5-7, 9
shifted: 7
should: 4, 27-28, 49,
57, 60, 65-67, 69-72,
74, 76, 80-81
showing: 48
signed: 4
similar: 10, 13, 18-19,

24, 37, 41, 43
simple: 6, 36-37, 60,
68-70, 76-77
simplest: 11, 17
simply: 11, 74
simulator: 60
single: 3, 9, 19, 74
sitstill: 43
situations: 39
smaller: 4
societies: 6
software: 4, 10-11,
18-19, 33, 35
solidity: 28, 31-32,
36, 39-45, 57,
59-60, 68, 74, 78
solution: 9-10, 44
solving: 4
someone: 5, 7
something: 41, 69, 71
sometimes: 5, 16
source: 32-33, 36, 40, 46
souvenir: 37
speaking: 35
special: 2, 12, 43
specific: 11, 35, 41, 75
specified: 76, 79-80
specifies: 75, 79
specify: 69, 77, 80
sphere: 4-5
spread: 3
standard: 35
started: 2
starting: 27
starts: 26
startups: 37
statically: 32
status: 20, 27
statuses: 26
storage: 11, 41
stored: 3, 11, 41-42,
44, 74-77, 80

stores: 35, 44
storing: 44
strategy: 17
string: 41, 45, 72-73, 75-80
strings: 5
strong: 12
struct: 42-45
structure: 2
structured: 24
students: 27
studio: 37, 40
stylesheet: 70
subject: 3
submit: 18, 51
suboptimal: 17
summary: 28, 50, 57, 81
supply: 20, 36-37, 55-56
support: 17
supports: 32
supposed: 45
surpasses: 75
symbols: 5
symmetric: 7-9, 28
synonyms: 12
syntax: 32, 40
system: 2-3, 6, 8, 12,
17, 27, 32-33, 37, 39,
44, 46-47, 57, 74
systems: 2-3, 6-7, 9, 17, 37

T

tbody: 70
tables: 8
taking: 19
target: 44
tasked: 65
techniques: 8
technology: 2, 4-5,
9, 24, 28
temporary: 42
tokens: 4, 20, 23, 27, 34,

37-38, 44, 49, 57
tolerance: 16-17
traitorous: 17
transfer: 38, 44, 55
transfers: 44
transform: 5
translates: 36
transmit: 6, 9
trials: 80
trigger: 74
turned: 8
twitter: 15
typical: 17
typically: 5

U

unable: 27
underlies: 16
underlying: 3, 5
understand: 28,
57, 60, 72, 81
uniformly: 3
unique: 3, 24
usually: 19, 33

V

validate: 3, 28
values: 20-21, 41, 74-80
variable: 20, 38, 40,
42, 45, 74-75
version: 3, 11-12, 14,
27, 37, 61-62
versions: 61
viewer: 26
visible: 35
visual: 37, 40
voters: 68
voting: 17, 60, 68-71

W

wallet: 1, 11-12, 14-15,
18-19, 27-28, 34-35,
47, 49, 54-56

warrior: 20

websites: 27

