Narayan Prusty

# Building Blockchain Projects

Develop real-time practical DApps using Ethereum and JavaScript

Packt

# Building Blockchain Projects

Develop real-time practical DApps using Ethereum and JavaScript

**Narayan Prusty**

BIRMINGHAM - MUMBAI

# Building Blockchain Projects

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book 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 book.

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

First published: April 2017

Production reference: 1240417

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78712-214-7

www.packtpub.com

# Credits

**Author**

Narayan Prusty

**Reviewers**

Imran Bahsir
Daniel Kraft
Gaurang Torvekar

**Commissioning Editor**

Veena Pagare

**Acquisition Editor**

Vinay Argekar

**Content Development Editor**

Mayur Pawanikar

**Technical Editor**

Prasad Ramesh

**Copy Editor**

Stuti Srivastava

**Project Coordinator**

Nidhi Joshi

**Proofreader**

Safis Editing

**Indexer**

Pratik Shirodkar

**Graphics**

Tania Dutta

**Production Coordinator**

Melwyn Dsa

# About the Author

**Narayan Prusty** is a full-stack developer, with five years of experience in the field. He specializes in Blockchain and JavaScript. His commitment has led him to build scalable products for startups, the government, and enterprises across India, Singapore, USA, and UAE.

At present, Ethereum, Bitcoin, Hyperledger, IPFS, Ripple, and so on are some of the things he uses on a regular basis to build decentralized applications. Currently, he is a full-time Blockchain SME (Subject-Matter Expert) at Emirates National Bank of Dubai.

He has already written two books on JavaScript titled *Learning ECMAScript 6* and *Modern JavaScript Applications*. Both these books were reviewed and published by Packt.

He starts working on something immediately if he feels it’s exciting and solves real work problems. He built an MP3 search engine at the age of 18, and since then, he has built various other applications, which are used by people around the globe. His ability to to build scalable applications from top to bottom is what makes him special.

Currently, he is on a mission to make things easier, faster, and cheaper using the blockchain technology. Also, he is looking at possibilities to prevent corruptions, fraud, and to bring transparency to the world using blockchain technology.
You can learn more from him from his blog `http://qnimate.com` and you can reach him out at LinkedIn `https://www.linkedin.com/in/narayanprusty/`.

# About the Reviewers

**Imran Bashir** has an M.Sc. degree in Information Security from Royal Holloway, University of London, and has a background in software development, solution architecture, infrastructure management, and IT service management. He is also a member of the Institute of Electrical and Electronics Engineers (IEEE) and the British Computer Society (BCS). Imran has sixteen years of experience in public and financial sector. He had worked on large-scale IT projects for the public sector before moving to the financial services industry. Since then, he worked in various technical roles for different financial companies in Europe's financial capital, London. He is currently working for an investment bank in London as Vice President in the technology department.

**Daniel Kraft** has studied mathematics and physics and holds a PhD degree in applied mathematics from the University of Graz in Austria. He has been involved in development with cryptocurrencies since 2013, has been the lead developer and chief scientist for both Namecoin and Huntercoin since 2014, and has published two research papers about cryptocurrency in peer-reviewed journals. He works as a software engineer and is a cofounder of Crypto Realities Ltd, a start-up that works on building decentralized multiplayer game worlds with blockchain technology.

**Gaurang Torvekar** has a master's degree in Information Systems from Singapore Management University. He is the cofounder and CTO of Attores, a Smart Contracts as a Service company, based in Singapore. He has extensive experience in Ethereum and Hyperledger application development. He has been a speaker at several blockchain conferences, conducted many hands on blockchain courses in Polytechnics in Singapore, and is also a Blockchain mentor at Angelhack.

# Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/178712214X`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

# Table of Contents

# Preface

Blockchain is a decentralized ledger that maintains a continuously growing list of data records secured from tampering and revision. Every user is allowed to connect to the network, send new transactions to it, verify transactions, and create new blocks.

This book will teach you what Blockchain is, how it maintains data integrity, and how to create real-world Blockchain projects using Ethereum. With interesting real-world projects, you will know learn how to write smart contracts which run exactly as programmed without any chance of fraud, censorship or third-party interference, and build end-to-end applications for Blockchain. You will learn concepts such as cryptography in cryptocurrencies, ether security, mining, smart contracts, and solidity.

The blockchain is the main technical innovation of bitcoin, where it serves as the public ledger for bitcoin transactions.

## What this book covers

`Chapter 1`, *Understanding Decentralized Applications*, will explain what DApps are and provide an an overview of how they work.

`Chapter 2`, *Understanding How Ethereum Works*, explains how Ethereum works.

`Chapter 3`, *Writing Smart Contracts*, shows how to write smart contracts and use geth's interactive console to deploy and broadcast transactions using web3.js.

`Chapter 4`, *Getting Started with web3.js*, introduces web3js and how to import, connect to geth, and explains use it in Node.js or client-side JavaScript.

`Chapter 5`, *Building a Wallet Service*, explains how to build a wallet service that users can create and manage Ethereum Wallets easily, even offline. We will specifically use the LightWallet library to achieve this.

`Chapter 6`, *Building a Smart Contract Deployment Platform*, shows how to compile smart contracts using web3.js and deploy it using web3.js and EthereumJS.

`Chapter 7`, *Building a Betting App*, explains how to use Oraclize to make HTTP requests from Ethereum smart contracts to access data from World Wide Web. We will also learn how to access files stored in IPFS, use the strings library to work with strings, and more.

`Chapter 8`, *Building Enterprise Level Smart Contracts*, explains how to use Truffle, which makes it easy to build enterprise-level DApps. We will learn about Truffle by building an alt-coin.

`Chapter 9`, *Building a Consortium Blockchain*, we will discuss consortium blockchain.

# What you need for this book

You require Windows 7 SP1+, 8, 10 or Mac OS X 10.8+.

# Who this book is for

This book is for JavaScript developers who now want to create tamper-proof data (and transaction) applications using Blockchain and Ethereum. Those who are interested in cryptocurrencies and the logic and database empowering it will find this book extremely useful.

# Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Then, run the app using the `node app.js` command inside the `Final` directory."

A block of code is set as follows:

```
var solc = require("solc");
var input = "contract x { function g() {} }";
var output = solc.compile(input, 1); // 1 activates the optimizer
for (var contractName in output.contracts) {
    // logging code and ABI
    console.log(contractName + ": " +
output.contracts[contractName].bytecode);
    console.log(contractName + "; " +
JSON.parse(output.contracts[contractName].interface));
}
```

Any command-line input or output is written as follows:

```
npm install –g solc
```

**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: "Now select the same file again and click on the **Get Info** button."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

# Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.
To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.
If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

# Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support`and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Building-Blockchain-Projects`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

## Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/BuildingBlockchainProjects_ColorImages.pdf`.

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support`and enter the name of the book in the search field. The required information will appear under the **Errata** section.

## Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

## Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

# 1
# Understanding Decentralized Applications

Almost all of the Internet-based applications we have been using are centralized, that is, the servers of each application are owned by a particular company or person. Developers have been building centralized applications and users have been using them for a pretty long time. But there are a few concerns with centralized applications that make it next to impossible to build certain types of apps and every app ends up having some common issues. Some issues with centralized apps are that they are less transparent, they have a single point of failure, they fail to prevent net censorship, and so on. Due to these concerns, a new technology emerged for the building of Internet-based apps called **decentralized applications** (**DApps**). In this chapter, we will learn about decentralized apps.

In this chapter, we'll cover the following topics:

- What are DApps?
- What is the difference between decentralized, centralized, and distributed applications?
- Advantages and disadvantages of centralized and decentralized applications.
- An overview of the data structures, algorithms, and protocols used by some of the most popular DApps
- Learning about some popular DApps that are built on top of other DApps.

# What is a DApp?

A DApp is a kind of Internet application whose backend runs on a decentralized peer-to-peer network and its source code is open source. No single node in the network has complete control over the DApp.

Depending on the functionality of the DApp, different data structures are used to store application data. For example, the Bitcoin DApp uses the blockchain data structure.

These peers can be any computer connected to the Internet; therefore, it becomes a big challenge to detect and prevent peers from making invalid changes to the application data and sharing wrong information with others. So we need some sort of consensus between the peers regarding whether the data published by a peer is right or wrong. There is no central server in a DApp to coordinate the peers and decide what is right and wrong; therefore, it becomes really difficult to solve this challenge. There are certain protocols (specifically called consensus protocols) to tackle this challenge. Consensus protocols are designed specifically for the type of data structure the DApp uses. For example, Bitcoin uses the proof-of-work protocol to achieve consensus.

Every DApp needs a client for the user to use the DApp. To use a DApp, we first need a node in the network by running our own node server of the DApp and then connecting the client to the node server. Nodes of a DApp provide an API only and let the developer community develop various clients using the API. Some DApp developers officially provide a client. Clients of DApps should be open source and should be downloaded for use; otherwise, the whole idea of decentralization will fail.

But this architecture of a client is cumbersome to set up, especially if the user is a non-developer; therefore, clients are usually hosted and/or nodes are hosted as a service to make the process of using a DApp easier.

**What are distributed applications?**
Distributed applications are those applications that are spread across multiple servers instead of just one. This is necessary when application data and traffic becomes huge and application downtime is not affordable. In distributed applications, data is replicated among various servers to achieve high availability of data. Centralized applications may or may not be distributed, but decentralized applications are always distributed. For example, Google, Facebook, Slack, Dropbox, and so on are distributed, whereas a simple portfolio site or a personal blog are not usually distributed until traffic is very high.

## Advantages of decentralized applications

Here are some of the advantages of decentralized applications:

- DApps are fault-tolerant as there is no single point of failure because they are distributed by default.
- They prevent violation of net censorship as there is no central authority to whom the government can pressurize to remove some content. Governments cannot even block the app's domain or IP address as DApps are not accessed via a particular IP address or domain. Obviously the government can track individual nodes in the network by their IP address and shut them down, but if the network is huge, then it becomes next to impossible to shut down the app, especially if the nodes are distributed among various different countries.
- It is easy for users to trust the application as it's not controlled by a single authority that could possibly cheat the users for profit.

## Disadvantages of decentralized applications

Obviously, every system has some advantages and disadvantages. Here are some of the disadvantages of decentralized applications:

- Fixing bugs or updating DApps is difficult, as every peer in the network has to update their node software.
- Some applications require verification of user identity (that is, KYC), and as there is no central authority to verify the user identity, it becomes an issue while developing such applications.
- They are difficult to build because they use very complex protocols to achieve consensus and they have to be built to scale from the start itself. So we cannot just implement an idea and then later on add more features and scale it.
- Applications are usually independent of third-party APIs to get or store something. DApps shouldn't depend on centralized application APIs, but DApps can be dependent on other DApps. As there isn't a large ecosystem of DApps yet, it is difficult to build a DApp. Although DApps can be dependent on other DApps theoretically, it is very difficult to tightly couple DApps practically.

# Decentralized autonomous organization

Typically, signed papers represent organizations, and the government has influence over them. Depending on the type of organization, the organization may or may not have shareholders.

**Decentralized autonomous organization** (**DAO**) is an organization that is represented by a computer program (that is, the organization runs according to the rules written in the program), is completely transparent, and has total shareholder control and no influence of the government.

To achieve these goals, we need to develop a DAO as a DApp. Therefore, we can say that DAO is a subclass of DApp.

Dash, and the DAC are a few example of DAOs.

**What is a decentralized autonomous corporation (DAC)?**
There is still no clear difference between DAC and DAO. Many people consider them to be the same whereas some people define DAC as DAO when DAO is intended to make profits for shareholders.

# User identity in DApps

One of the major advantages of DApps is that it generally guarantees user anonymity. But many applications require the process of verifying user identity to use the app. As there is no central authority in a DApp, it become a challenge to verify the user identity.

In centralized applications, humans verify user identity by requesting the user to submit certain scanned documents, OTP verification, and so on. This process is called **know your customer** (**KYC**). But as there is no human to verify user identity in DApps, the DApp has to verify the user identity itself. Obviously DApps cannot understand and verify scanned documents, nor can they send SMSes; therefore, we need to feed them with digital identities that they can understand and verify. The major problem is that hardly any DApps have digital identities and only a few people know how to get a digital identity.

There are various forms of digital identities. Currently, the most recommended and popular form is a digital certificate. A digital certificate (also called a public key certificate or identity certificate) is an electronic document used to prove ownership of a public key. Basically, a user owns a private key, public key, and digital certificate. The private key is secret and the user shouldn't share it with anyone. The public key can be shared with anyone. The digital certificate holds the public key and information about who owns the public key. Obviously, it's not difficult to produce this kind of certificate; therefore, a digital certificate is always issued by an authorized entity that you can trust. The digital certificate has an encrypted field that's encrypted by the private key of the certificate authority. To verify the authenticity of the certificate, we just need to decrypt the field using the public key of the certificate authority, and if it decrypts successfully, then we know that the certificate is valid.

Even if users successfully get digital identities and they are verified by the DApp, there is a still a major issue; that is, there are various digital certificate issuing authorities, and to verify a digital certificate, we need the public key of the issuing authority. It is really difficult to include the public keys of all the authorities and update/add new ones. Due to this issue, the procedure of digital identity verification is usually included on the client side so that it can be easily updated. Just moving this verification procedure to the client side doesn't completely solve this issue because there are lots of authorities issuing digital certificates and keeping track of all of them, and adding them to the client side, is cumbersome.

TIP

**Why do users not verify each other's identity?**
Often, while we do trading in real life, we usually verify the identity of the other person ourselves or we bring in an authority to verify the identity. This idea can be applied to DApps as well. Users can verify each other's identity manually before performing trade with each other. This idea works for specific kinds of DApps, that is, for DApps in which people trade with each other. For example, if a DApp is a decentralized social network, then obviously a profile cannot be verified by this means. But if the DApp is for people to buy/sell something, then before making a payment, the buyer and seller can both verify each other's identity. Although this idea may seem fine while doing trading, when you think practically, it becomes very difficult because you may not want to do identity verification every time you trade and everyone not knows how to do identity verification. For example, if the DApp is a cab-booking app, then you will obviously not want to perform identity verification before booking a cab every time. But if you trade sometimes and you know how to verify identity, then it's fine to follow this procedure.

Due to these issues, the only option we are currently left with is verifying user identity manually by an authorized person of the company that provides the client. For example, to create a Bitcoin account, we don't need an identification, but while withdrawing Bitcoin to flat currency, the exchanges ask for proof of identification. Clients can omit the unverified users and not let them use the client. And they can keep the client open for users whose identity has been verified by them. This solution also ends up with minor issues; that is, if you switch the client, you will not find the same set of users to interact with because different clients have different sets of verified users. Due to this, all users may decide to use a particular client only, thus creating a monopoly among clients. But this isn't a major issue because if the client fails to properly verify users, then users can easily move to another client without losing their critical data, as they are stored as decentralized.

The idea of verifying user identity in applications is to make it difficult for users to escape after performing some sort of fraudulent activity, preventing users with a fraud/criminal background from using the application, and providing the means for other users in the network to believe a user to be whom the user is claiming to be. It doesn't matter what procedure is used to verify user identity; they are always ways for users to represent themselves to be someone else. It doesn't matter whether we use digital identities or scanned documents for verification because both can be stolen and reused. What's important is just to make it difficult for users to represent themselves to be someone else and also collect enough data to track a user and prove that the user has done a fraudulent activity.

# User accounts in DApps

Many applications need user accounts' functionality. Data associated with an account should be modifiable by the account owner only. DApps simply cannot have the same username- and password-based account functionality as do centralized applications because passwords cannot prove that the data change for an account has been requested by the owner.

There are quite a few ways to implement user accounts in DApps. But the most popular way is using a public-private key pair to represent an account. The hash of the public key is the unique identifier of the account. To make a change to the account's data, the user needs to sign the change using his/her private key. We need to assume that users will store their private keys safely. If users lose their private keys, then they lose access to their account forever.

# Accessing the centralized apps

A DApp shouldn't depend on centralized apps because of a single point of failure. But in some cases, there is no other option. For example, if a DApp wants to read a football score, then where will it get the data from? Although a DApp can depend on another DApp, why will FIFA create a DApp? FIFA will not create a DApp just because other DApps want the data. This is because a DApp to provide scores is of no benefit as it will ultimately be controlled by FIFA completely.

So in some cases, a DApp needs to fetch data from a centralized application. But the major problem is how the DApp knows that the data fetched from a domain is not tampered by a middle service/man and is the actual response. Well, there are various ways to resolve this depending on the DApp architecture. For example, in Ethereum, for the smart contracts to access centralized APIs, they can use the Oraclize service as a middleman as smart contracts cannot make direct HTTP requests. Oraclize provides a TLSNotary proof for the data it fetches for the smart contract from centralized services.

# Internal currency in DApps

For a centralized application to sustain for a long time, the owner of the app needs to make a profit in order to keep it running. DApps don't have an owner, but still, like any other centralized app, the nodes of a DApp need hardware and network resources to keep it running. So the nodes of a DApp need something useful in return to keep the DApp running. That's where internal currency comes into play. Most DApps have a built-in internal currency, or we can say that most successful DApps have a built-in internal currency.

The consensus protocol is what decides how much currency a node receives. Depending on the consensus protocol, only certain kinds of nodes earn currency. We can also say that the nodes that contribute to keeping the DApp secure and running are the ones that earn currency. Nodes that only read data are not rewarded with anything. For example, in Bitcoin, only miners earn Bitcoins for successfully mining blocks.

The biggest question is since this is a digital currency, why would someone value it? Well, according to economics, anything that has demand and whose supply is insufficient will have value.

Making users pay to use the DApp using the internal currency solves the demand problem. As more and more users use the DApp, the demand also increases and, therefore, the value of the internal currency increases as well.

Setting a fixed amount of currency that can be produced makes the currency scarce, giving it a higher value.

The currency is supplied over time instead of supplying all the currency at a go. This is done so that new nodes that enter the network to keep it secure and running also earn the currency.

## Disadvantages of internal currency in DApps

The only demerit of having internal currency in DApps is that the DApps are not free for use anymore. This is one of the places where centralized applications get the upper hand as centralized applications can be monetized using ads, providing premium APIs for third-party apps, and so and can be made free for users.

In DApps, we cannot integrate ads because there is no one to check the advertising standards; the clients may not display ads because there is no benefit for them in displaying ads.

## What are permissioned DApps?

Until now, we have been learning about DApps, which are completely open and permissionless; that is, anyone can participate without establishing an identity.

On the other hand, permissioned DApps are not open for everyone to participate. Permissioned DApps inherit all properties of permissionless DApps, except that you need permission to participate in the network. Permission systems vary between permissioned DApps.

To join a permissioned DApp, you need permission, so consensus protocols of permissionless DApps may not work very well in permissioned DApps; therefore, they have different consensus protocols than permissionless DApps. Permissioned DApps don't have internal currency.

# Popular DApps

Now that we have some high-level knowledge about what DApps are and how they are different from centralized apps, let's explore some of the popular and useful DApps. While exploring these DApps, we will explore them at a level that is enough to understand how they work and tackle various issues instead of diving too deep.

# Bitcoin

Bitcoin is a decentralized currency. Bitcoin is the most popular DApp and its success is what showed how powerful DApps can be and encouraged people to build other DApps.

Before we get into further details about how Bitcoin works and why people and the government consider it to be a currency, we need to learn what ledgers and blockchains are.

## What is a ledger?

A ledger is basically a list of transactions. A database is different from a ledger. In a ledger, we can only append new transactions, whereas in a database, we can append, modify, and delete transactions. A database can be used to implement a ledger.

## What is blockchain?

A blockchain is a data structure used to create a decentralized ledger. A blockchain is composed of blocks in a serialized manner. A block contains a set of transactions, a hash of the previous block, timestamp (indicating when the block was created), block reward, block number, and so on. Every block contains a hash of the previous block, thus creating a chain of blocks linked with each other. Every node in the network holds a copy of the blockchain.

Proof-of-work, proof-of-stake, and so on are various consensus protocols used to keep the blockchain secure. Depending on the consensus protocol, the blocks are created and added to the blockchain differently. In proof-of-work, blocks are created by a procedure called mining, which keeps the blockchain safe. In the proof-of-work protocol, mining involves solving complex puzzles. We will learn more about blockchain and its consensus protocols later in this book.

The blockchain in the Bitcoin network holds Bitcoin transactions. Bitcoins are supplied to the network by rewarding new Bitcoins to the nodes that successfully mine blocks.

The major advantage of blockchain data structure is that it automates auditing and makes an application transparent yet secure. It can prevent fraud and corruption. It can be used to solve many other problems depending on how you implement and use it.

## Is Bitcoin legal?

First of all, Bitcoin is not an internal currency; rather, it's a decentralized currency. Internal currencies are mostly legal because they are an asset and their use is obvious.

The main question is whether currency-only DApps are legal or not. The straight answer is that it's legal in many countries. Very few countries have made it illegal and most are yet to decide.

Here are a few reasons why some countries have made it illegal and most are yet to decide:

- Due to the identity issue in DApps, user accounts don't have any identity associated with them in Bitcoin; therefore, it can be used for money laundering
- These virtual currencies are very volatile, so there is a higher risk of people losing money
- It is really easy to evade taxes when using virtual currencies

## Why would someone use Bitcoin?

The Bitcoin network is used to only send/receive Bitcoins and nothing else. So you must be wondering why there would be demand for Bitcoin.

Here are some reasons why people use Bitcoin:

- The major advantage of using Bitcoin is that it makes sending and receiving payments anywhere in the world easy and fast
- Online payment transaction fees are expensive compared to Bitcoin transaction fees
- Hackers can steal your payment information from merchants, but in the case of Bitcoin, stealing Bitcoin addresses is completely useless because for a transaction to be valid, it must be signed with its associated private key, which the user doesn't need to share with anyone to make a payment.

# Ethereum

Ethereum is a decentralized platform that allows us to run DApps on top of it. These DApps are written using smart contracts. One or more smart contracts can form a DApp together. An Ethereum smart contract is a program that runs on Ethereum. A smart contract runs exactly as programmed without any possibility of downtime, censorship, fraud, and third-party interference.

The main advantage of using Ethereum to run smart contracts is that it makes it easy for smart contracts to interact with each other. Also, you don't have to worry about integrating consensus protocol and other things; instead, you just need to write the application logic. Obviously, you cannot build any kind of DApp using Ethereum; you can build only those kinds of DApps whose features are supported by Ethereum.

Ethereum has an internal currency called ether. To deploy smart contracts or execute functions of the smart contracts, you need ether.

This book is dedicated to building DApps using Ethereum. Throughout this book, you will learn every bit of Ethereum in depth.

# The Hyperledger project

Hyperledger is a project dedicated to building technologies to build permissioned DApps. Hyperledger fabric (or simply fabric) is an implementation of the Hyperledger project. Other implementations include Intel Sawtooth and R3 Corda.

Fabric is a permissioned decentralized platform that allows us to run permissioned DApps (called chaincodes) on top of it. We need to deploy our own instance of fabric and then deploy our permissioned DApps on top of it. Every node in the network runs an instance of fabric. Fabric is a plug-and-play system where you can easily plug and play various consensus protocols and features.

Hyperledger uses the blockchain data structure. Hyperledger-based blockchains can currently choose to have no consensus protocols (that is, the **NoOps** protocol) or else use the **PBFT** (**Practical Byzantine Fault Tolerance**) consensus protocol. It has a special node called certificate authority, which controls who can join the network and what they can do.

# IPFS

**IPFS** (**InterPlanetary File System**) is a decentralized filesystem. IPFS uses **DHT** (**distributed hash table**) and Merkle **DAG** (**directed acyclic graph**) data structures. It uses a protocol similar to BitTorrent to decide how to move data around the network. One of the advanced features of IPFS is that it supports file versioning. To achieve file versioning, it uses data structures similar to Git.

Although it called a decentralized filesystem, it doesn't adhere to a major property of a filesystem; that is, when we store something in a filesystem, it is guaranteed to be there until deleted. But IPFS doesn't work that way. Every node doesn't hold all files; it stores the files it needs. Therefore, if a file is less popular, then obviously many nodes won't have it; therefore, there is a huge chance of the file disappearing from the network. Due to this, many people prefer to call IPFS a decentralized peer-to-peer file-sharing application. Or else, you can think of IPFS as BitTorrent, which is completely decentralized; that is, it doesn't have a tracker and has some advanced features.

## How does it work?

Let's look at an overview of how IPFS works. When we store a file in IPFS, it's split into chunks < 256 KB and hashes of each of these chunks are generated. Nodes in the network hold the IPFS files they need and their hashes in a hash table.

There are four types of IPFS files: blob, list, tree, and commit. A blob represents a chunk of an actual file that's stored in IPFS. A list represents a complete file as it holds the list of blobs and other lists. As lists can hold other lists, it helps in data compression over the network. A tree represents a directory as it holds a list of blobs, lists, other trees, and commits. And a commit file represents a snapshot in the version history of any other file. As lists, trees, and commits have links to other IPFS files, they form a Merkle DAG.

So when we want to download a file from the network, we just need the hash of the IPFS list file. Or if we want to download a directory, then we just need the hash of the IPFS tree file.

As every file is identified by a hash, the names are not easy to remember. If we update a file, then we need to share a new hash with everyone that wants to download that file. To tackle this issue, IPFS uses the IPNS feature, which allows IPFS files to be pointed using self-certified names or human-friendly names.

### Filecoin

The major reason that is stopping IPFS from becoming a decentralized filesystem is that nodes only store the files they need. Filecoin is a decentralized filesystem similar to IPFS with an internal currency to incentivize nodes to store files, thus increasing file availability and making it more like a filesystem.

Nodes in the network will earn Filecoins to rent disk space, and to store/retrieve files, you need to spend Filecoins.

Along with IPFS technologies, Filecoin uses the blockchain data structure and the proof-of-retrievability consensus protocol.

At the time of writing this, Filecoin is still under development, so many things are still unclear.

# Namecoin

Namecoin is a decentralized key-value database. It has an internal currency too, called Namecoins. Namecoin uses the blockchain data structure and the proof-of-work consensus protocol.

In Namecoin, you can store key-value pairs of data. To register a key-value pair, you need to spend Namecoins. Once you register, you need to update it once in every 35,999 blocks; otherwise, the value associated with the key will expire. To update, you need Namecoins as well. There is no need to renew the keys; that is, you don't need to spend any Namecoins to keep the key after you have registered it.

Namecoin has a namespace feature that allows users to organize different kinds of keys. Anyone can create namespaces or use existing ones to organize keys.

Some of the most popular namespaces are `a` (application specific data), `d` (domain name specifications), `ds` (secure domain name), `id` (identity), `is` (secure identity), `p` (product), and so on.

## .bit domains

To access a website, a browser first finds the IP address associated with the domain. These domain name and IP address mappings are stored in DNS servers, which are controlled by large companies and governments. Therefore, domain names are prone to censorship. Governments and companies usually block domain names if the website is doing something illegal or making loss for them or due to some other reason.

Due to this, there was a need for a decentralized domain name database. As Namecoin stores key-value data just like DNS servers, Namecoin can be used to implement a decentralized DNS, and this is what it has already been used for. The d and ds namespaces contain keys ending with `.bit`, representing `.bit` domain names. Technically, a namespace doesn't have any naming convention for the keys but all the nodes and clients of Namecoin agree to this naming convention. If we try to store invalid keys in `d` and `ds` namespaces, then clients will filter invalid keys.

A browser that supports `.bit` domains needs to look up in the Namecoin's `d` and `ds` namespace to find the IP address associated with the `.bit` domain.

The difference between the `d` and `ds` namespaces is that `ds` stores domains that support TLS and `d` stores the ones that don't support TLS. We have made DNS decentralized; similarly, we can also make the issuing of TLS certificates decentralized.

This is how TLS works in Namecoin. Users create self-signed certificates and store the certificate hash in Namecoin. When a client that supports TLS for `.bit` domains tries to access a secured `.bit` domain, it will match the hash of the certificate returned by the server with the hash stored in Namecoin, and if they match, then they proceed with further communication with the server.

> A decentralized DNS formed using Namecoin is the first solution to the Zooko triangle. The Zooko triangle defines applications that have three properties, that is, decentralized, identity, and secure. Digital identity is used not only to represent a person, but it can also represent a domain, company, or something else.

# Dash

Dash is a decentralized currency similar to Bitcoin. Dash uses the blockchain data structure and the proof-of-work consensus protocol. Dash solves some of the major issues that are caused by Bitcoin. Here are some issues related to Bitcoin:

- Transactions take a few minutes to complete, and in today's world, we need transactions to complete instantly. This is because the mining difficulty in the Bitcoin network is adjusted in such a way that a block gets created once in an average of every 10 minutes. We will learn more about mining later on in this book.
- Although accounts don't have an identity associated with them, trading Bitcoins for real currency on an exchange or buying stuff with Bitcoins is traceable; therefore, these exchanges or merchants can reveal your identity to governments or other authorities. If you are running your own node to send/receive transactions, then your ISP can see the Bitcoin address and trace the owner using the IP address because broadcasted messages in the Bitcoin network are not encrypted.

Dash aims to solve these problems by making transactions settle almost instantly and making it impossible to identify the real person behind an account. It also prevents your ISP from tracking you.

In the Bitcoin network, there are two kinds of nodes, that is, miners and ordinary nodes. But in Dash, there are three kinds of nodes, that is, miners, masternodes, and ordinary nodes. Masternodes are what makes Dash so special.

## Decentralized governance and budgeting

To host a masternode, you need to have 1,000 Dashes and a static IP address. In the Dash network, both masternodes and miners earn Dashes. When a block is mined, 45% reward goes to the miner, 45% goes to the masternodes, and 10% is reserved for the budget system.

Masternodes enable decentralized governance and budgeting. Due to the decentralized governance and budgeting system, Dash is called a DAO because that's exactly what it is.

Masternodes in the network act like shareholders; that is, they have rights to take decisions regarding where the 10% Dash goes. This 10% Dash is usually used to funds other projects. Each masternode is given the ability to use one vote to approve a project.

Discussions on project proposals happen out of the network. But the voting happens in the network.

> Masternodes can provide a possible solution to verify user identity in DApps; that is, masternodes can democratically select a node to verify user identity. The person or business behind this node can manually verify user documents. A part of this reward can also go to this node. If the node doesn't provide good service, then the masternodes can vote for a different node. This can be a fine solution to the decentralized identity issue.

## Decentralized service

Instead of just approving or rejecting a proposal, masternodes also form a service layer that provides various services. The reason that masternodes provide services is that the more services they provide, the more feature-rich the network becomes, thus increasing users and transactions, which increases prices for Dash currency and the block reward also gets high, therefore helping masternodes earn more profit.

Masternodes provide services such as PrivateSend (a coin-mixing service that provides anonymity), InstantSend (a service that provides almost instant transactions), DAPI (a service that provides a decentralized API so that users don't need to run a node), and so on.

At a given time, only 10 masternodes are selected. The selection algorithm uses the current block hash to select the masternodes. Then, we request a service from them. The response that's received from the majority of nodes is said to be the correct one. This is how consensus is achieved for services provided by the masternodes.

The proof-of-service consensus protocol is used to make sure that the masternodes are online, are responding, and have their blockchain up-to-date.

# BigChainDB

BigChainDB allows you to deploy your own permissioned or permissionless decentralized database. It uses the blockchain data structure along with various other database-specific data structures. BigChainDB, at the time of writing this, is still under development, so many things are not clear yet.

It also provides many other features, such as rich permissions, querying, linear scaling, and native support for multi-assets and the federation consensus protocol.

# OpenBazaar

OpenBazaar is a decentralized e-commerce platform. You can buy or sell goods using OpenBazaar. Users are not anonymous in the OpenBazaar network as their IP address is recorded. A node can be a buyer, seller, or a moderator.

It uses a Kademlia-style distributed hash table data structure. A seller must host a node and keep it running in order to make the items visible in the network.

It prevents account spam by using the proof-of-work consensus protocol. It prevents ratings and reviews spam using proof-of-burn, CHECKLOCKTIMEVERIFY, and security deposit consensus protocols.

Buyers and sellers trade using Bitcoins. A buyer can add a moderator while making a purchase. The moderator is responsible for resolving a dispute if anything happens between the buyer and the seller. Anyone can be a moderator in the network. Moderators earn commission by resolving disputes.

# Ripple

Ripple is decentralized remittance platform. It lets us transfer fiat currencies, digital currencies, and commodities. It uses the blockchain data structure and has its own consensus protocol. In ripple docs, you will not find the term blocks and blockchain; they use the term ledger instead.

In ripple, money and commodity transfer happens via a trust chain in a manner similar to how it happens in a hawala network. In ripple, there are two kinds of nodes, that is, gateways and regular nodes. Gateways support deposit and withdrawal of one or more currencies and/or commodities. To become a gateway in a ripple network, you need permission as gateways to form a trust chain. Gateways are usually registered financial institutions, exchanges, merchants, and so on.

Every user and gateway has an account address. Every user needs to add a list of gateways they trust by adding the gateway addresses to the trust list. There is no consensus to find whom to trust; it all depends on the user, and the user takes the risk of trusting a gateway. Even gateways can add the list of gateways they trust.

Let's look at an example of how user X living in India can send 500 USD to user Y living in the USA. Assuming that there is a gateway XX in India, which takes cash (physical cash or card payments on their website) and gives you only the INR balance on ripple, X will visit the XX office or website and deposit 30,000 INR and then XX will broadcast a transaction saying I owe X 30,000 INR. Now assume that there is a gateway YY in the USA, which allows only USD transactions and Y trusts YY gateway. Now, say, gateways XX and YY don't trust each other. As X and Y don't trust a common gateway, XX and YY don't trust each other, and finally, XX and YY don't support the same currency. Therefore, for X to send money to Y, he needs to find intermediary gateways to form a trust chain. Assume there is another gateway, ZZ, that is trusted by both XX and YY and it supports USD and INR. So now X can send a transaction by transferring 50,000 INR from XX to ZZ and it gets converted to USD by ZZ and then ZZ sends the money to YY, asking YY to give the money to Y. Now instead of X owing Y $500, YY owes $500 to Y, ZZ owes $500 to YY, and XX owes 30,000 INR to ZZ. But it's all fine because they trust each other, whereas earlier, X and Y didn't trust each other. But XX, YY, and ZZ can transfer the money outside of ripple whenever they want to, or else a reverse transaction will deduct this value.

Ripple also has an internal currency called XRP (or ripples). Every transaction sent to the network costs some ripples. As XRP is the ripple's native currency, it can be sent to anyone in the network without trust. XRP can also be used while forming a trust chain. Remember that every gateway has its own currency exchange rate. XRP isn't generated by a mining process; instead, there are total of 100 billion XRPs generated in the beginning and owned by the ripple company itself. XRP is supplied manually depending on various factors.

All the transactions are recorded in the decentralized ledger, which forms an immutable history. Consensus is required to make sure that all nodes have the same ledger at a given point of time. In ripple, there is a third kind of node called validators, which are part of the consensus protocol. Validators are responsible for validating transactions. Anyone can become a validator. But other nodes keep a list of validators that can be actually trusted. This list is known as UNL (unique node list). A validator also has a UNL; that is, the validators it trusts as validators also want to reach a consensus. Currently, ripple decides the list of validators that can be trusted, but if the network thinks that validators selected by ripple are not trustworthy, then they can modify the list in their node software.

You can form a ledger by taking the previous ledger and applying all the transactions that have happened since then. So to agree on the current ledger, nodes must agree on the previous ledger and the set of transactions that have happened since then. After a new ledger is created, a node (both regular nodes and validators) starts a timer (of a few seconds, approximately 5 seconds) and collects the new transactions that arrived during the creation of the previous ledger. When the timer expires, it takes those transactions that are valid according to at least 80% of the UNLs and forms the next ledger. Validators broadcast a proposal (a set of transactions they think are valid to form the next ledger) to the network. Validators can broadcast proposals for the same ledger multiple times with a different set of transactions if they decide to change the list of valid transactions depending on proposals from their UNLs and other factors. So you only need to wait 5-10 seconds for your transaction to be confirmed by the network.

Some people wonder whether this can lead to many different versions of the ledger since each node may have a different UNL. As long as there is a minimal degree of inter-connectivity between UNLs, a consensus will rapidly be reached. This is primarily because every honest node's primary goal is to achieve a consensus.

# Summary

In this chapter, we learned what DApps are and got an an overview of how they work. We looked at some of the challenges faced by DApps and the various solutions to these issues. Finally, we saw some of the popular DApps and had an overview of what makes them special and how they work. Now you should be comfortable explaining what a DApp is and how it works.

# 2
# Understanding How Ethereum Works

In the previous chapter, we saw what DApps are. We also saw an overview of some of the popular DApps. One of them was Ethereum. At present, Ethereum is the most popular DApp after bitcoin. In this chapter, we will learn in depth about how Ethereum works and what we can develop using Ethereum. We will also see the important Ethereum clients and node implementations.

In this chapter, we will cover the following topics:

- Ethereum user accounts
- What are smart contracts and how do they work?
- Ethereum virtual machine
- How does mining work in the proof-of-work consensus protocol?
- Learning how to use the geth command
- Setting up the Ethereum Wallet and Mist
- Overview of Whisper and Swarm
- The future of Ethereum

# Overview of Ethereum

Ethereum is a decentralized platform, which allows us to deploy DApps on top of it. Smart contracts are written using the solidity programming language. DApps are created using one or more smart contracts. Smart contracts are programs that run exactly as programmed without any possibility of downtime, censorship, fraud, or third party interface. In Ethereum, smart contracts can be written in several programming languages, including Solidity, LLL, and Serpent. Solidity is the most popular of those languages. Ethereum has an internal currency called ether. To deploy smart contracts or to call their methods, we need ether. There can be multiple instances of a smart contract just like any other DApp, and each instance is identified by its unique address. Both user accounts and smart contracts can hold ether.

Ethereum uses blockchain data structure and proof-of-work consensus protocol. A method of a smart contract can be invoked via a transaction or via another method. There are two kinds of nodes in the network: regular nodes and miners. Regular nodes are the ones that just have a copy of the blockchain, whereas miners build the blockchain by mining blocks.

# Ethereum accounts

To create an Ethereum account, we just need an asymmetric key pair. There are various algorithms, such as RSA, ECC, and so on, for generating asymmetric encryption keys. Ethereum uses **elliptic curve cryptography** (**ECC**). ECC has various parameters. These parameters are used to adjust speed and security. Ethereum uses the `secp256k1` parameter. To go in depth about ECC and its parameters will require mathematical knowledge, and it's not necessary to understand it in depth for building DApps using Ethereum.

Ethereum uses 256-bit encryption. An Ethereum private/public key is a 256-bit number. As processors cannot represent such big numbers, it's encoded as a hexadecimal string of length 64.

Every account is represented by an address. Once we have the keys we need to generate the address, here is the procedure to generate the address from the public key:

1. First, generate the `keccak-256` hash of the public key. It will give you a 256-bit number.
2. Drop the first 96 bits, that is, 12 bytes. You should now have 160 bits of binary data, that is, 20 bytes.
3. Now encode the address as a hexadecimal string. So finally, you will have a bytestring of 40 characters, which is your account address.

Now anyone can send ether to this address.

# Transactions

A **transaction** is a signed data package to transfer ether from an account to another account or to a contract, invoke methods of a contract, or deploy a new contract. A transaction is signed using **ECDSA** (**Elliptic Curve Digital Signature Algorithm**), which is a digital signature algorithm based on ECC. A transaction contains the recipient of the message, a signature identifying the sender and proving their intention, the amount of ether to transfer, the maximum number of computational steps the transaction execution is allowed to take (called the gas limit), and the cost the sender of the transaction is willing to pay for each computational step (called the gas price). If the transaction's intention is to invoke a method of a contract, it also contains input data, or if its intention is to deploy a contract, then it can contain the initialization code. The product of gas used and gas price is called transaction fees. To send ether or to execute a contract method, you need to broadcast a transaction to the network. The sender needs to sign the transaction with its private key.

A transaction is said to be confirmed if we are sure that it will always appear in the blockchain. It is recommended to wait for 15 confirmations before assuming a transaction to be confirmed.

# Consensus

Every node in the Ethereum network holds a copy of the blockchain. We need to make sure that nodes cannot tamper with the blockchain, and we also need a mechanism to check whether a block is valid or not. And also, if we encounter two different valid blockchains, we need to have a way to find out which one to choose.

Ethereum uses the proof-of-work consensus protocol to keep the blockchain tamper-proof. A proof-of-work system involves solving a complex puzzle to create a new block. Solving the puzzle should require a significant amount of computational power thereby making it difficult to create blocks. The process of creating blocks in the proof-of-work system is called mining. Miners are the nodes in the network that mine blocks. All the DApps that use proof-of-work do not implement exactly the same set of algorithms. They may differ in terms of what the puzzle miners need to solve, how difficult the puzzle is, how much time it takes to solve it, and so on. We will learn about proof-of-work with respect to Ethereum.

Anyone can become a miner in the network. Every miner solves the puzzle individually; the first miner to solve the puzzle is the winner and is rewarded with five ether and transaction fees of all the transactions in that block. If you have a more powerful processor than any other node in the network, that doesn't mean that you will always succeed because the parameters for the puzzle are not exactly same for all the miners. But instead, if you have a more powerful processor than any other node in the network, it gives you a higher chance at succeeding. Proof-of-work behaves like a lottery system, and processing power can be thought as the number of lottery tickets a person has. Networks security is not measured by total number of miners; instead, it's measured by the total processing power of the network.

There is no limit to the number of blocks the blockchain can have, and there is no limit to the total ether that can be produced. Once a miner successfully mines a block, it broadcasts the block to all other nodes in the network. A block has a header and a set of transactions. Every block holds hash of the previous block, thereby creating a connected chain.

Let's see what the puzzle the miners need to solve is and how it's solved at a high level. To mine a block, first of all, a miner collects the new un-mined transactions broadcasted to it, and then it filters out the not-valid transactions. A transaction to be valid must be properly signed using the private key, the account must have enough balance to make the transaction, and so on. Now the miner creates a block, which has a header and content. Content is the list of transactions that the block contains. The header contains things such as the hash of the previous block, block number, nonce, target, timestamp, difficulty, address of the miner, and so on. The timestamp represents the time at the block's inception. Then nonce is a meaningless value, which is adjusted in order to find the solution to the puzzle. The puzzle is basically to find such nonce values with which when the block is hashed, the hash is less than or equal to the target. Ethereum uses ethash hashing algorithm. The only way to find the nonce is to enumerate all possibilities. The target is a 256-bit number, which is calculated based on various factors. The difficulty value in the header is a different representation of the target to make it easier to deal with. The lower the target, the more time it takes to find the nonce, and the higher the target, the less time it takes to find the nonce. Here is the formula to calculate the difficulty of the puzzle:

```
current_block_difficulty = previous_block_difficulty +
previous_block_difficulty // 2048 * max(1 - (current_block_timestamp -
previous_blocktimestamp) // 10, -99) + int(2 ** ((current_block_number //
100000) - 2))
```

Now any node in the network can check whether the blockchain they have is valid or not by first checking whether the transactions in the blockchain are valid, the timestamp validation, then whether the target and nonce of all the blocks are valid, a miner has assigned a valid reward itself, and so on.

If a node in the network receives two different valid blockchains, then the blockchain whose combined difficulty of all blocks is higher is considered to be the valid blockchain.

Now, for example, if a node in the network alters some transactions in a block, then the node needs to calculate the nonce of all the succeeding blocks. By the time it re-finds the nonce of the succeeding blocks, the network would have mined many more blocks and therefore reject this blockchain as its combined difficulty would be lower.

## Timestamp

The formula to calculate the target of a block requires the current timestamp, and also every block has the current timestamp attached to its header. Nothing can stop a miner from using some other timestamp instead of the current timestamp while mining a new block, but they don't usually because timestamp validation would fail and other nodes won't accept the block, and it would be a waste of resources of the miner. When a miner broadcasts a newly mined block, its timestamp is validated by checking whether the timestamp is greater than the timestamp of the previous block. If a miner uses a timestamp greater than the current timestamp, the difficulty will be low as difficulty is inversely proportional to the current timestamp; therefore, the miner whose block timestamp is the current timestamp would be accepted by the network as it would have a higher difficulty. If a miner uses a timestamp greater than the previous block timestamp and less than the current timestamp, the difficulty would be higher, and therefore, it would take more time to mine the block; by the time the block is mined, the network would have produced more blocks, therefore, this block will get rejected as the blockchain of the malicious miner will have a lower difficulty than the blockchain the network has. Due to these reasons, miners always use accurate timestamps, otherwise they gain nothing.

## Nonce

The nonce is a 64-bit unsigned integer. The nonce is the solution to the puzzle. A miner keeps incrementing the nonce until it finds the solution. Now you must be wondering if there is a miner who has hash power more than any other miner in the network, would the miner always find nonce first? Well, it wouldn't.

The hash of the block that the miners are mining is different for every miner because the hash depends on things such as the timestamp, miner address, and so on, and it's unlikely that it will be the same for all miners. Therefore, it's not a race to solve the puzzle; rather, it's a lottery system. But of course, a miner is likely to get lucky depending on its hash power, but that doesn't mean the miner will always find the next block.

# Block time

The block difficulty formula we saw earlier uses a 10-second threshold to make sure that the difference between the time a parent and child block mines is in is between 10-20 seconds. But why is it 10-20 seconds and not some other value? And why there such a constant time difference restriction instead of a constant difficulty?

Imagine that we have a constant difficulty, and miners just need to find a nonce to get the hash of the block less and equal to the difficulty. Suppose the difficulty is high; then in this case, users will have no way to find out how long it will take to send ether to another user. It may take a very long time if the computational power of the network is not enough to find the nonce to satisfy the difficulty quickly. Sometimes the network may get lucky and find the nonce quickly. But this kind of system will find it difficult to gain attraction from users as users will always want to know how much time it should take for a transaction to be completed, just like when we transfer money from one bank account to another bank account, we are given a time period within which it should get completed. If the constant difficulty is low, it will harm the security of the blockchain because large miners can mine blocks much faster than small miners, and the largest miner in the network will have the ability to control the DApp. It is not possible to find a constant difficulty value that can make the network stable because the network's computational power is not constant.

Now we know why we should always have an average time for how long it should take for the network to mine a block. Now the question is what the most suitable average time is as it can be anything from 1 second to infinite seconds. A smaller average time can be achieved by lowering the difficulty, and higher average time can be achieved by increasing the difficulty. But what are the merits and demerits of a lower and higher average time? Before we discuss this, we need to first know what stale blocks are.

What happens if two miners mine the next block at nearly the same time? Both the blocks will be valid for sure, but the blockchain cannot hold two blocks with the same block number, and also, both the miners cannot be awarded. Although this is a common issue, the solution is simple. In the end, the blockchain with the higher difficulty will be the one accepted by the network. So the valid blocks that are finally left out are called stale blocks.

The total number of stale blocks produced in the network is inversely proportional to the average time it takes to generate a new block. Shorter block generation time means there would be less time for the newly mined block to propagate throughout the network and a bigger chance of more than one miner finding a solution to the puzzle, so by the time the block is propagated through the network, some other miner would have also solved the puzzle and broadcasted it, thereby creating stales. But if the average block generation time is bigger, there is less chance that multiple miners will be able to solve the puzzle, and even if they solve it, there is likely to be time gap between when they solved it, during which the first solved block can be propagated and the other miners can stop mining that block and proceed towards mining the next block. If stale blocks occur frequently in the network, they cause major issues, but if they occur rarely, they do no harm.

But what's the problem with stale blocks? Well, they delay the confirmation of a transaction. When two miners mine a block at nearly the same time, they may not have the same set of transactions, so if our transactions appears in one of them, we cannot say that it's confirmed as the block in which the transaction appeared may be stale. And we should wait for a few more blocks to be mined. Due to stale blocks, the average confirmation time is not equal to average block generation time.

Do stale blocks impact blockchain security? Yes, they do. We know that the network's security is measured by the total computation power of the miners in the network. When computation power increases, the difficulty is increased to make sure that blocks aren't generated earlier than the average block time. So more difficulty means a more secure blockchain, as for a node to tamper, the blockchain will need much more hash power now, which makes it more difficult to tamper with the blockchain; therefore, the blockchain is said to be more secure. When two blocks are mined at nearly the same time, we will have the network parted in two, working on two different blockchains, but one is going to be the final blockchain. So the part of the network working on the stale block mines the next block on top of the stale block, which ends up in loss of hash power of the network as hash power is being used for something unnecessary. The two parts of the network are likely to take longer than the average block time to mine the next block as they have lost hash power; therefore, after mining the next block, there will be decrease in difficulty as it took more time than the average block time to mine the block. The decrease in difficulty impacts the overall blockchain security. If the stale rate is too high, it will affect the blockchain security by a huge margin.

Ethereum tackles the security issue caused by stale blocks using something known as ghost protocol. Ethereum uses a modified version of the actual ghost protocol. The ghost protocol covers up the security issue by simply adding the stale blocks into the main blockchain, thereby increasing the overall difficulty of the blockchain, as overall difficulty of the blockchain also includes the sum of difficulties of the stale blocks. But how are stale blocks inserted into the main blockchain without transactions conflicting? Well, any block can specify 0 or more stales. To incentivize miners to include stale blocks, the miners are rewarded for including stale blocks. And also, the miners of the stale blocks are rewarded. The transactions in the stale blocks are not used for calculating confirmations, and also, the stale block miners don't receive the transaction fees of the transactions included in the stale blocks. Note that Ethereum calls stale blocks uncle blocks.

Here is the formula to calculate how much reward a miner of a stale block receives. The rest of the reward goes to the nephew block, that is, the block that includes the orphan block:

```
(uncle_block_number + 8 - block_number) * 5 / 8
```

As not rewarding the miners of stale blocks doesn't harm any security, you must be wondering why miners of stale blocks get rewarded? Well, there is another issue caused when stale blocks occur frequently in the network, which is solved by rewarding the miners of stale blocks. A miner should earn a percentage of reward similar to the percentage of hash power it contributes to the network. When a block is mined at nearly the same time by two different miners, then the block mined by the miner with more hash power is more likely to get added to the final blockchain because of the miner's efficiency to mine the next block; therefore, the small miner will lose reward. If the stale rate is low, it's not a big issue because the big miner will get a little increase in reward; but if the stale rate is high, it causes a big issue, that is, the big miner in the network will end up taking much more rewards than it should receive. The ghost protocol balances this by rewarding the miners of stale blocks. As the big miner doesn't take all the rewards but much more than it should get, we don't award stale block miners the same as the nephew block; instead, we award a lesser amount to balance it. The preceding formula balances it pretty well.

Ghost limits the total number of stale blocks a nephew can reference so that miners don't simply mine stale blocks and stall the blockchain.

So wherever a stale block appears in the network, it somewhat affects the network. The more the frequency of stale blocks, the more the network is affected by it.

# Forking

A fork is said to have happened when there is a conflict among the nodes regarding the validity of a blockchain, that is, more than one blockchain happens to be in the network, and every blockchain is validated for some miners. There are three kinds of forks: regular forks, soft fork, and hard fork.

A regular fork is a temporary conflict occurring due to two or more miners finding a block at nearly the same time. It's resolved when one of them has more difficulty than the other.

A change to the source code could cause conflicts. Depending on the type of conflict, it may require miners with more than 50% of hash power to upgrade or all miners to upgrade to resolve the conflict. When it requires miners with more than 50% of hash power to upgrade to resolve the conflict, its called a soft fork, whereas when it requires all the miners to upgrade to resolve the conflict, its called a hard fork. An example of a soft fork would be if update to the source code invalidates subset of old blocks/transactions, then it can be resolved when miners more than 50% of hash power have upgraded so that the new blockchain will have more difficulty and finally get accepted by the whole network. An example of a hard fork would be an if update in the source code was to change the rewards for miners, then all the miners needs to upgrade to resolve the conflict.

Ethereum has gone through various hard and soft forks since its release.

# Genesis block

A genesis block is the first block of the blockchain. It's assigned to block number 0. It's the only block in the blockchain that doesn't reference to a previous block because there isn't any. It doesn't hold any transactions because there isn't any ether produced yet.

Two nodes in a network will only pair with each other if they both have the same genesis block, that is, blocks synchronization will only happen if both peers have the same genesis block, otherwise they both will reject each other. A different genesis block of high difficulty cannot replace a lower difficult one. Every node generates its own genesis block. For various networks, the genesis block is hardcoded into the client.

# Ether denominations

Ether has various denominations just like any other currency. Here are the denominations:

- 1 Ether = 1000000000000000000 Wei
- 1 Ether = 1000000000000000 Kwei
- 1 Ether = 1000000000000 Mwei
- 1 Ether = 1000000000 Gwei
- 1 Ether = 1000000 Szabo
- 1 Ether = 1000 Finney
- 1 Ether = 0.001 Kether
- 1 Ether = 0.000001 Mether
- 1 Ether = 0.000000001 Gether
- 1 Ether = 0.000000000001 Tether

# Ethereum virtual machine

EVM (or Ethereum virtual machine) is the Ethereum smart contracts byte-code execution environment. Every node in the network runs EVM. All the nodes execute all the transactions that point to smart contracts using EVM, so every node does the same calculations and stores the same values. Transactions that only transfer ether also require some calculation, that is, to find out whether the address has a balance or not and deduct the balance accordingly.

Every node executes the transactions and stores the final state due to various reasons. For example, if there is a smart contract that stores the names and details of everyone attending a party, whenever a new person is added, a new transaction is broadcasted to the network. For any node in the network to display details of everyone attending the party, they simply need to read the final state of the contract.

Every transaction requires some computation and storage in the network. Therefore, there needs to be a transaction cost, otherwise the whole network will be flooded with spam transactions, and also without a transaction cost, miners will have no reason to include transactions in blocks, and they will start mining empty blocks. Every transaction requires different amount of computation and storage; therefore, every transaction has different transaction costs.

There are two implementations of EVM, that is, byte-code VM and JIT-VM. At the time of writing this book, JIT-VM is available for use, but its development is still not completed. In either case, the Solidity code is compiled to byte code. In the case of JIT-VM, the byte code is further compiled. JIT-VM is more efficient than it's counterpart.

# Gas

Gas is a unit of measurement for computational steps. Every transaction is required to include a gas limit and a fee that it is willing to pay per gas (that is, pay per computation); miners have the choice of including the transaction and collecting the fee. If the gas used by the transaction is less than or equal to the gas limit, the transaction processes. If the total gas exceeds the gas limit, then all changes are reverted, except that the transaction is still valid and the fee (that is, the product of the maximum gas that can be used and gas price) can still be collected by the miner.

The miners decide the gas price (that is, price per computation). If a transaction has a lower gas price than the gas price decided by a miner, the miner will refuse to mine the transaction. The gas price is an amount in a wei unit. So, a miner can refuse to include a transaction in a block if the gas price is lower than what it needs.

Each operation in EVM is assigned a number of how much gas it consumes.

Transaction costs affect the maximum ether an account can transfer to another account. For example, if an account has an ether balance of five, it cannot transfer all five ethers to another account because if all ethers are transferred, there would be no balance in the account to deduct transaction fees from.

If a transaction invokes a contract method and the method sends some ether or invokes some other contract method, the transaction fee is deducted from the account that invoked the contract method.

# Peer discovery

For a node to be part of the network, it needs to connect to some other nodes in the network so that it can broadcast transactions/blocks and listen to new transactions/blocks. A node doesn't need to connect to every node in the network; instead, a node connects to a few other nodes. And these nodes connect to a few other nodes. In this way, the whole network is connected to each other.

But how does a node find some other nodes in the network as there is no central server that everyone can connect to so as to exchange their information? Ethereum has its own node discovery protocol to solve this problem, which is based on the Kadelima protocol. In the node discovery protocol, we have special kind of nodes called Bootstrap nodes. Bootstrap nodes maintain a list of all nodes that are connected to them over a period of time. They don't hold the blockchain itself. When peers connect to the Ethereum network, they first connect to the Bootstrap nodes ,which share the lists of peers that have connected to them in the last predefined time period. The connecting peers then connect and synchronize with the peers.

There can be various Ethereum instances, that is, various networks, each having its own network ID. The two major Ethereum networks are mainnet and testnet. The mainnet one is the one whose ether is traded on exchanges, whereas testnet is used by developers to test. Until now, we have learned everything with regards to the mainnet blockchain.

Bootnode is the most popular implementation of an Ethereum Bootstrap node. If you want to host your own Bootstrap node, you can use bootnode.

# Whisper and Swarm

Whisper and Swarm are a decentralized communication protocol and a decentralized storage platform respectively, being developed by Ethereum developers. Whisper is a decentralized communication protocol, whereas Swarm is a decentralized filesystem.

Whisper lets nodes in the network communicate with each other. It supports broadcasting, user-to-user, encrypted messages, and so on. It's not designed to transfer bulk data. You can learn more about Whisper at `https://github.com/ethereum/wiki/wiki/Whisper`, and you can see a code example overview at `https://github.com/ethereum/wiki/wiki/Whisper-Overview`.

Swarm is similar to Filecoin, that is, it differs mostly in terms of technicalities and incentives. Filecoin doesn't penalize stores, whereas Swarm penalizes stores; therefore, this increases the file availability further. You must be wondering how incentive works in swarm. Does it have an internal currency? Actually, Swarm doesn't have an internal currency, rather it uses ether for incentives. There is a smart contract in Ethereum, which keeps track of incentives. Obviously, the smart contract cannot communicate with Swarm; instead, swarm communicates with the smart contract. So basically, you pay the stores via the smart contract, and the payment is released to the stores after the expiry date. You can also report file missing to the smart contract, in which case it can penalize the respective stores. You can learn more about the difference between Swarm and IPFS/Filecoin at `https://github.com/ethersphere/go-ethereum/wiki/IPFS-&-SWARM` and see the smart contract code at `https://github.com/ethersphere/go-ethereum/blob/bzz-config/bzz/bzzcontract/swarm.sol`.

At the time of writing this book, Whisper and Swarm are still under development; so, many things are still not clear.

# Geth

Geth (or called as go-ethereum) is an implementation of Ethereum, Whisper, and Swarm nodes. Geth can be used to be part of all of these or only selected ones. The reason for combining them is to make them look like a single DApp and also so that via one node, a client can access all three DApps.

Geth is a CLI application. It's written in the go programming language. It's available for all the major operating systems. The current version of geth doesn't yet support Swarm and supports whisper a some of the features of Whisper. At the time of writing this book, the latest version of geth was 1.3.5.

# Installing geth

Geth is available for OS X, Linux, and Windows. It supports two types of installation: binary and scripted installation. At the time of writing this book, the latest stable version of geth is 1.4.13. Let's see how to install it in various operating systems using the binary installation method. Scripted installation is used when you have to modify something in the geth source code and install it. We don't want to make any changes to the source code, therefore, we will go with binary installation.

## OS X

The recommended way of installing geth in OS X is using brew. Run these two commands in the terminal to install geth:

```
brew tap ethereum/ethereum
brew install ethereum
```

## Ubuntu

The recommended way to install geth in Ubuntu is to use `apt-get`. Run these commands in Ubuntu terminal to install geth:

```
sudo apt-get install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install ethereum
```

## Windows

Geth comes as an executable file for Windows. Download the zip file from `https://github.com/ethereum/go-ethereum/wiki/Installation-instructions-for-Windows`, and extract it. Inside it, you will find the `geth.exe` file.

To find more about installing geth on various operating systems, visit `https://github.com/ethereum/go-ethereum/wiki/Building-Ethereum`.

# JSON-RPC and JavaScript console

Geth provides JSON-RPC APIs for other applications to communicate with it. Geth serves JSON-RPC APIs using HTTP, WebSocket, and other protocols. The APIs provided by JSON-RPC are divided into these categories: admin, debug, eth, miner, net, personal, shh, txpool, and web3. You can find more information about it these `https://github.com/ethereum/go-ethereum/wiki/JavaScript-Console`.

Geth also provides an interactive JavaScript console to interact with it programmatically using JavaScript APIs. This interactive console uses JSON-RPC over IPC to communicate with geth. We will learn more about the JSON-RPC and the JavaScript APIs in later chapters.

# Sub-commands and options

Let's learn some of the important sub-commands and options of the geth command using examples. You can find the list of all sub-commands and options by using the help sub-command. We will see a lot more about geth and its commands throughout the following chapters.

## Connecting to the mainnet network

Nodes in the Ethereum network, by default, communicate using `30303` port. But nodes are also free to listen on some other port numbers.

To connect to the mainnet network, you just need to run the `geth` command. Here is an example of how to specify the network ID explicitly and specify a custom directory where geth will store the downloaded blockchain:

```
    geth --datadir "/users/packt/ethereum" --networkid 1
```

The `--datadir` option is used to specify where to store the blockchain. If it's not provided, the default path is `$HOME/.ethereum`.

`--networkid` is used to specify the network ID. 1 is the ID of the mainnet network. If it's not provided, the default value is 1. The network ID of testnet is 2.

## Creating a private network

To create a private network, you just need to give a random network ID. Private networks are usually created for development purposes. Geth also provide various flags related to logging and debugging, which are useful during development. So, instead of giving a random network ID and putting the various logging and debugging flags, we can simply use the `--dev` flag, which runs a private network with various debugging and logging flags enabled.

# Creating accounts

Geth also lets us create accounts, that is, generate keys and addresses associated with them. To create an account, use the following command:

```
    geth account new
```

When you run this command, you will be asked to enter a password to encrypt your account. If you forget your password, there is no way to access your account.

To get a list of all accounts in your local Wallet, use the following command:

```
        geth account list
```

The preceding command will print a list of all the addresses of the accounts. Keys are, by default, stored in the `--datadir` path, but you can use the `--keystore` option to specify a different directory.

## Mining

By default, geth doesn't start mining. To instruct geth to start mining, you just need to provide the `--mine` option. There are a few other options related to mining:

```
        geth --mine --minerthreads 16 --minergpus '0,1,2' --etherbase
    '489b4e22aab35053ecd393b9f9c35f4f1de7b194' --unlock
    '489b4e22aab35053ecd393b9f9c35f4f1de7b194'
```

Here, along with the `--mine` option, we have provided various other options. The `--minerthreads` option specifies the total number of threads to use while hashing. By default, eight threads are used. Etherbase is the address to which the reward earned by mining is deposited. By default, accounts are encrypted. So to access the ether in the account, we need to unlock it, that is, decrypt the account. Decryption is used to decrypt the private key associated with the account. To start mining, we don't need to unlock it because only the address is required to deposit the mining rewards. One or more accounts can be unlocked using the -unlock option. Multiple addresses can be provided by separating the addresses using comma.

`--minergpus` is used to specify the GPUs to use for mining. To get the list of GPUs, use the `geth gpuinfo` command. For each GPU, you need to have 1-2 GB of RAM. By default, it doesn't use GPUs, instead only CPU.

## Fast synchronization

At the time of writing this book, the blockchain size is around 30 GB. Downloading it may take several hours or days if you have a slow Internet connection. Ethereum implements a fast synchronization algorithm, which can download the blockchain faster.

Fast synchronization doesn't download the entire blocks; instead, it only downloads the block headers, transactions receipts, and the recent state database. So, we don't have to download and replay all transactions. To check blockchain integrity, the algorithm downloads a full block after every defined number of blocks. To learn more about fast synchronization algorithm, visit `https://github.com/ethereum/go-ethereum/pull/1889`.

To use fast sync while downloading the blockchain, you need to use the `--fast` flag while running geth.

Due to security reasons, fast sync will only run during an initial sync (that is, when the node's own blockchain is empty). After a node manages to successfully sync with the network, fast sync is forever disabled. As an additional safety feature, if a fast sync fails close to or after the random pivot point, it is disabled as a safety precaution, and the node reverts to full, block-processing-based synchronization.

# Ethereum Wallet

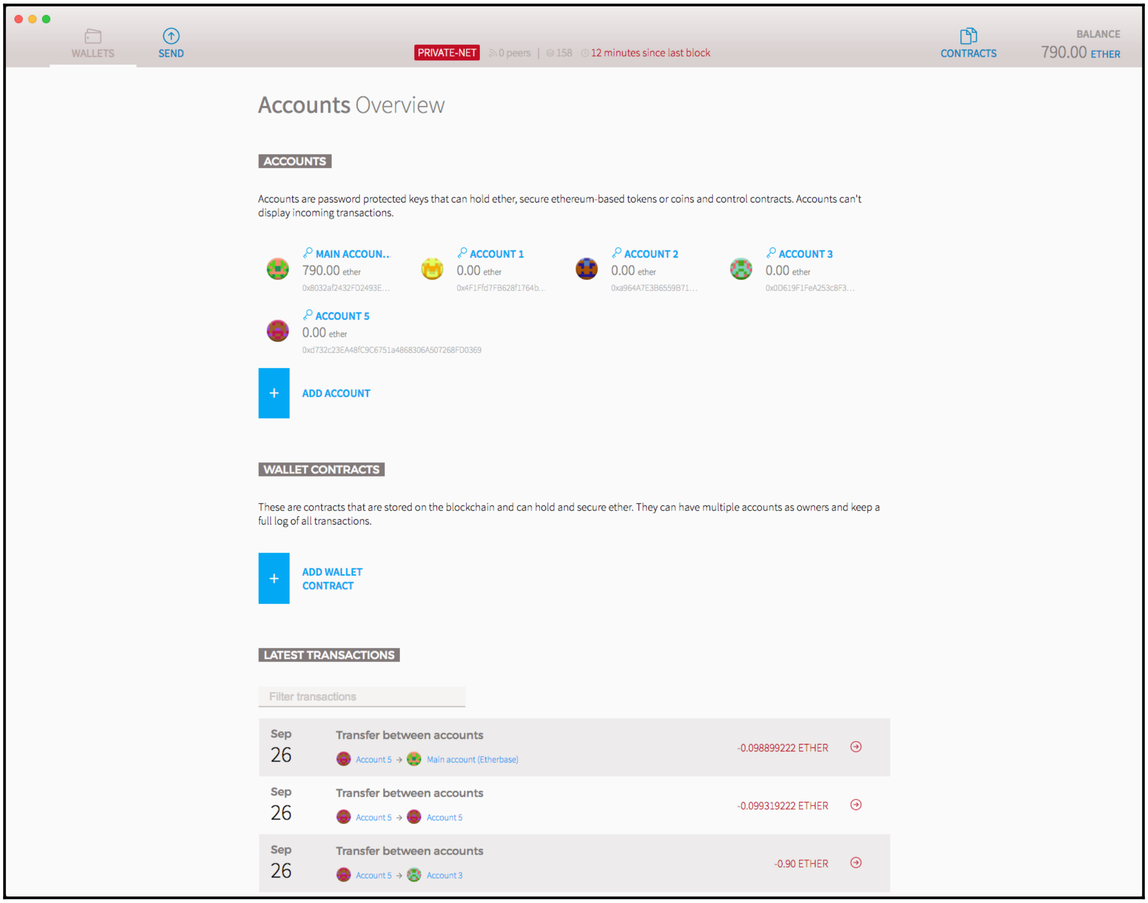
Ethereum Wallet is an Ethereum UI client that lets you create account, send ether, deploy contracts, invoke methods of contracts, and much more.

Ethereum Wallet comes with geth bundled. When you run Ethereum, it tries to find a local geth instance and connects to it, and if it cannot find geth running, it launches its own geth node. Ethereum Wallet communicates with geth using IPC. Geth supports file-based IPC.

> If you change the data directory while running geth, you are also changing the IPC file path. So for Ethereum Wallet to find and connect to your geth instance, you need to use the `--ipcpath` option to specify the IPC file location to its default location so that Ethereum Wallet can find it; otherwise Ethereum Wallet won't be able to find it and will start its own geth instance. To find the default IPC file path, run geth help, and it will show the default path next to the `--ipcpath` option.

Visit `https://github.com/ethereum/mist/releases` to download Ethereum Wallet. It's available for Linux, OS X, and Windows. Just like geth, it has two installation modes: binary and scripted installation.

Here is an image that shows what Ethereum Wallet looks like:



Accounts Overview

ACCOUNTS

Accounts are password protected keys that can hold ether, secure ethereum-based tokens or coins and control contracts. Accounts can't display incoming transactions.

MAIN ACCOUN..
790.00 ether
0x8032af2432FD2493E...

ACCOUNT 1
0.00 ether
0x4F1Ffd7FB628f1764b...

ACCOUNT 2
0.00 ether
0xa964A7E3B6559B71...

ACCOUNT 3
0.00 ether
0x0D619F1FeA253c8F3...

ACCOUNT 5
0.00 ether
0xd732c23EA48fC9C6751a4868306A507268FD0369

WALLET CONTRACTS

These are contracts that are stored on the blockchain and can hold and secure ether. They can have multiple accounts as owners and keep a full log of all transactions.

LATEST TRANSACTIONS

Sep 26 Transfer between accounts
Account 5 → Main account (Etherbase) -0.098899222 ETHER

Sep 26 Transfer between accounts
Account 5 → Account 5 -0.099319222 ETHER

Sep 26 Transfer between accounts
Account 5 → Account 3 -0.90 ETHER

# Mist

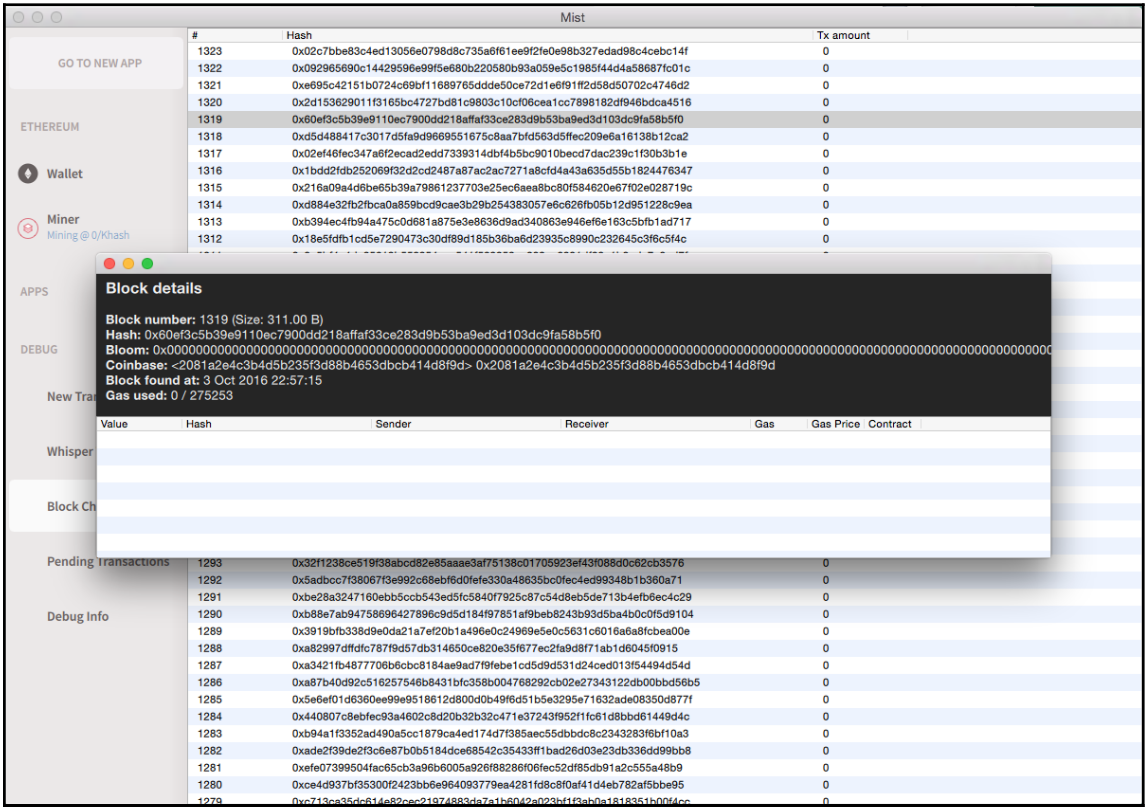
Mist is a client for Ethereum, Whisper, and Swarm. It lets us send transactions, send Whisper messages, inspect blockchains, and so on.

The relation between Mist and geth is similar to the relation between Ethereum Wallet and geth.

The most popular feature of Mist is that it comes with a browser. Currently, the frontend JavaScript running in the browser can access the web3 APIs of the geth node using the `web3.js` library (a library that provides Ethereum console's JavaScript APIs for other applications to communicate with geth).

The basic idea of Mist is to build the third generation web (Web 3.0), which would wipe out the need to have servers by using Ethereum, Whisper, and Swarm as replacements for centralized servers.

Here is an image, showing what Mist looks like:



| # | Hash | Tx amount |
|---|---|---|
| 1323 | 0x02c7bbe83c4ed13056e0798d8c735a6f61ee9f2fe0e98b327edad98c4cebc14f | 0 |
| 1322 | 0x092965690c14429596e99f5e680b220580b93a059e5c1985f44d4a58687fc01c | 0 |
| 1321 | 0xe695c42151b0724c69bf11689765ddde50ce72d1e6f91ff2d58d50702c4746d2 | 0 |
| 1320 | 0x2d153629011f3165bc4727bd81c9803c10cf06cea1cc7898182df946bdca4516 | 0 |
| 1319 | 0x60ef3c5b39e9110ec7900dd218affaf33ce283d9b53ba9ed3d103dc9fa58b5f0 | 0 |
| 1318 | 0xd5d488417c3017d5fa9d9669551675c8aa7bfd563d5ffec209e6a16138b12ca2 | 0 |
| 1317 | 0x02ef46fec347a6f2ecad2edd7339314dbf4b5bc9010becd7dac239c1f30b3b1e | 0 |
| 1316 | 0x1bdd2fdb252069f32d2cd2487a87ac2ac7271a8cfd4a43a635d55b1824476347 | 0 |
| 1315 | 0x216a09a4d6be65b39a79861237703e25ec6aea8bc80f584620e67f02e028719c | 0 |
| 1314 | 0xd884e32fb2fbca0a859bcd9cae3b29b254383057e6c626fb05b12d951228c9ea | 0 |
| 1313 | 0xb394ec4fb94a475c0d681a875e3e8636d9ad340863e946ef6e163c5bfb1ad717 | 0 |
| 1312 | 0x18e5fdfb1cd5e7290473c30df89d185b36ba6d23935c8990c232645c3f6c5f4c | 0 |
| 1293 | 0x32f1238ce519f38abcd82e85aaae3af75138c01705923ef43f088d0c62cb3576 | 0 |
| 1292 | 0x5adbcc7f38067f3e992c68ebf6d0fefe330a48635bc0fec4ed99348b1b360a71 | 0 |
| 1291 | 0xbe28a3247160ebb5ccb543ed5fc5840f7925c87c54d8eb5de713b4efb6ec4c29 | 0 |
| 1290 | 0xb88e7ab94758696427896c9d5d184f97851af9beb8243b93d5ba4b0c0f5d9104 | 0 |
| 1289 | 0x3919bfb338d9e0da21a7ef20b1a496e0c24969e5e0c5631c6016a6a8fcbea00e | 0 |
| 1288 | 0xa82997dffdfc787f9d57db314650ce820e35f677ec2fa9d8f71ab1d6045f0915 | 0 |
| 1287 | 0xa3421fb4877706b6cbc8184ae9ad7f9febe1cd5d9d531d24ced013f54494d54d | 0 |
| 1286 | 0xa87b40d92c516257546b8431bfc358b004768292cb02e27343122db00bbd56b5 | 0 |
| 1285 | 0x5e6ef01d6360ee99e9518612d800d0b49f6d51b5e3295e71632ade08350d877f | 0 |
| 1284 | 0x440807c8ebfec93a4602c8d20b32b32c471e37243f952f1fc61d8bbd61449d4c | 0 |
| 1283 | 0xb94a1f3352ad490a5cc1879ca4ed174d7f385aec55dbbdc8c2343283f6bf10a3 | 0 |
| 1282 | 0xade2f39de2f3c6e87b0b5184dce68542c35433ff1bad26d03e23db336dd99bb8 | 0 |
| 1281 | 0xefe07399504fac65cb3a96b6005a926f88286f06fec52df85db91a2c555a48b9 | 0 |
| 1280 | 0xce4d937bf35300f2423bb6e964093779ea4281fd8c8f0af41d4eb782af5bbe95 | 0 |
| 1279 | 0xc713ca35dc614e82cec21974883da7a1b6042a023bf1f3ab0a1818351b00f4cc | 0 |

**Block details**

**Block number:** 1319 (Size: 311.00 B)
**Hash:** 0x60ef3c5b39e9110ec7900dd218affaf33ce283d9b53ba9ed3d103dc9fa58b5f0
**Bloom:** 0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
**Coinbase:** <2081a2e4c3b4d5b235f3d88b4653dbcb414d8f9d> 0x2081a2e4c3b4d5b235f3d88b4653dbcb414d8f9d
**Block found at:** 3 Oct 2016 22:57:15
**Gas used:** 0 / 275253

| Value | Hash | Sender | Receiver | Gas | Gas Price | Contract |
|---|---|---|---|---|---|---|

# Weaknesses

Every system has some weaknesses. Similarly, Ethereum also has some weaknesses. Obviously, just like any other application, Ethereum source code can have bugs. And also just like any other network-based application, Ethereum is also exposed to DoS attacks. But let's see the unique and most important weaknesses of Ethereum.

## Sybil attack

An attacker can attempt to fill the network with regular nodes controlled by him; you would then be very likely to connect only to the attacker nodes. Once you have connected to the attacker nodes, the attacker can refuse to relay blocks and transactions from everyone, thereby disconnecting you from the network. The attacker can relay only blocks that he creates, thereby putting you on a separate network, and so on.

## 51% attack

If the attacker controls more than half of the network hashrate, the attacker can generate blocks faster than the rest of the network. The attacker can simply preserve his private fork until it becomes longer than the branch built by the honest network and then broadcast it.

With more than 50% of hash power, the miner can reverse transactions, prevent all/some transactions from getting mined, and prevent other miners' mined blocks from getting inserted to the blockchain.

# Serenity

Serenity is the name of the next major update for Ethereum. At the time of writing this book, serenity is still under development. This update will require a hard fork. Serenity will change the consensus protocol to casper, and will integrate state channels and sharding. Complete details of how these will work is still unclear at this point of time. Let's see a high level overview of what these are.

# Payment and state channels

Before getting into state channels, we need to know what payment channels are. A payment channel is a feature that allows us to combine more than two transactions of sending ether to another account into two transactions. Here is how it works. Suppose X is the owner of a video streaming website, and Y is a user. X charges one ether for every minute. Now X wants Y to pay after every minute while watching the video. Of course, Y can broadcast a transaction every minute, but there are few issues here, such as X has to wait for confirmation, so the video will be paused for sometime, and so on. This is the problem payment channels solve. Using payment channels, Y can lock some ether (maybe 100 ether) for a period of time (maybe 24 hours) for X by broadcasting a lock transaction. Now after watching a 1 minute video, Y will send a signed record indicating that the lock can be unlocked and one ether will go to X's account and the rest to Y's account. After another minute, Y will send a signed record indicating that the lock can be unlocked, and two ether will go to X's account, and the rest will go to Y's account. This process will keep going as Y watches the video on X's website. Now once Y has watched 100 hours of video or 24 hours of time is about to be reached, X will broadcast the final signed record to the network to withdraw funds to his account. If X fails to withdraw in 24 hours, the complete refund is made to Y. So in the blockchain, we will see only two transactions: lock and unlock.

Payment channel is for transactions related to sending ether. Similarly, a state channel allows us to combine transactions related to smart contracts.

# Proof-of-stake and casper

Before we get into what the casper consensus protocol is, we need to understand how the proof-of-stake consensus protocol works.

Proof-of-stake is the most common alternative to proof-of-work. Proof-of-work wastes too many computational resources. The difference between POW and POS is that in POS, a miner doesn't need to solve the puzzle; instead the miner needs to prove ownership of the stake to mine the block. In the POS system, ether in accounts is treated as a stake, and the probability of a miner mining the block is directly proportional to the stake the miner holds. So if the miner holds 10% of the stake in the network, it will mine 10% of the blocks.

But the question is how will we know who will mine the next block? We cannot simply let the miner with the highest stake always mine the next block because this will create centralization. There are various algorithms for next block selection, such as randomized block selection, and coin-age-based selection.

Casper is a modified version of POS that tackles various problems of POS.

# Sharding

At present, every node needs to download all transactions, which is huge. At the rate at which blockchain size is increasing, in the next few years, it will be very difficult to download the whole blockchain and keep it in sync.

If you are familiar with distributed database architecture, you must be familiar with sharding. If not, then sharding is a method of distributing data across multiple computers. Ethereum will implement sharding to partition and distribute the blockchain across nodes.

You can learn more about sharding a blockchain at `https://github.com/ethereum/wiki/wiki/Sharding-FAQ`.

# Summary

In this chapter, we learned in detail about how Ethereum works. We learned how block time affects security and about the weaknesses of Ethereum. We also saw what Mist and Ethereum Wallet are and how to install them. We also saw some of the important commands of geth. Finally, we learned what is going to be new in Serenity updates for Ethereum.

In the next chapter, we will learn about the various ways to store and protect ether.

# 3
# Writing Smart Contracts

In the previous chapter, we learned how the Ethereum blockchain works and how the PoW consensus protocol keeps it safe. Now it's time to start writing smart contracts as we have have a good grasp of how Ethereum works. There are various languages to write Ethereum smart contracts in, but Solidity is the most popular one. In this chapter, we will learn the Solidity programming language. We will finally build a DApp for proof of existence, integrity, and ownership at given a time, that is, a DApp that can prove that a file was with a particular owner at a specific time.

In this chapter, we'll cover the following topics:

- The layout of Solidity source files
- Understanding Solidity data types
- Special variables and functions of contracts
- Control structures
- Structure and features of contracts
- Compiling and deploying contracts

## Solidity source files

A Solidity source file is indicated using the `.sol` extension. Just like any other programming language, there are various versions of Solidity. The latest version at the time of writing this book is 0.4.2.

In the source file, you can mention the compiler version for which the code is written for using the `pragma Solidity` directive.

For example, take a look at the following:

```
pragma Solidity ^0.4.2;
```

Now the source file will not compile with a compiler earlier than version 0.4.2, and it will also not work on a compiler starting from version 0.5.0 (this second condition is added using ^). Compiler versions between 0.4.2 to 0.5.0 are most likely to include bug fixes instead of breaking anything.

It is possible to specify much more complex rules for the compiler version; the expression follows those used by npm.

# The structure of a smart contract

A contract is like a class. A contract contains state variables, functions, function modifiers, events, structures, and enums. Contracts also support inheritance. Inheritance is implemented by copying code at the time of compiling. Smart contracts also support polymorphism.

Let's look at an example of a smart contract to get an idea about what it looks like:

```
contract Sample
{
    //state variables
    uint256 data;
    address owner;

    //event definition
    event logData(uint256 dataToLog);

    //function modifier
    modifier onlyOwner() {
        if (msg.sender != owner) throw;
        _;
    }

    //constructor
    function Sample(uint256 initData, address initOwner){
        data = initData;
        owner = initOwner;
    }

    //functions
```

```
        function getData() returns (uint256 returnedData){
            return data;
        }

        function setData(uint256 newData) onlyOwner{
            logData(newData);
            data = newData;
        }
    }
```

Here is how the preceding code works:

- At first, we declared a contract using the `contract` keyword.
- Then, we declared two state variables; `data` holds some data and owner holds the Ethereum wallet address of the owner, that is, the address in which the contract was deployed.
- Then, we defined an event. Events are used to notify the client about something. We will trigger this event whenever `data` changes. All events are kept in the blockchain.
- Then, we defined a function modifier. Modifiers are used to automatically check a condition prior to executing a function. Here, the modifier checks whether the owner of the contract is invoking the function or not. If not, then it throws an exception.
- Then, we have the contract constructor. While deploying the contract, the constructor is invoked. The constructor is used to initialize the state variables.
- Then, we defined two methods. The first method was to get the value of the `data` state variable and the second was a method to change the `data` value.

Before getting any further deeper into the features of smart contracts, let's learn some other important things related to Solidity. And then we will come back to contracts.

# Data location

All programming languages you would have learned so far store their variables in memory. But in Solidity, variables are stored in the memory and the filesystem depending on the context.

Depending on the context, there is always a default location. But for complex data types, such as strings, arrays, and structs, it can be overridden by appending either `storage` or `memory` to the type. The default for function parameters (including return parameters) is memory, the default for local variables is storage. and the location is forced to storage, for state variables (obviously).

Data locations are important because they change how assignments behave:

- Assignments between storage variables and memory variables always create an independent copy. But assignments from one memory-stored complex type to another memory-stored complex type do not create a copy.
- Assignment to a state variable (even from other state variables) always creates an independent copy.
- You cannot assign complex types stored in memory to local storage variables.
- In case of assigning state variables to local storage variables, the local storage variables point to the state variables; that is, local storage variables become pointers.

# What are the different data types?

Solidity is a statically typed language; the type of data a variable holds needs to be predefined. By default, all bits of the variables are assigned to 0. In Solidity, variables are function scoped; that is, a variable declared anywhere within a function will be in scope for the entire function regardless of where it is declared.

Now let's look at the various data types provided by Solidity:

- The most simple data type is `bool`. It can hold either `true` or `false`.
- `uint8`, `uint16`, `uint24` ... `uint256` are used to hold unsigned integers of 8 bits, 16 bits, 24 bits ... 256 bits, respectively. Similarly, `int8`, `int16` ... `int256` are used to hold signed integers of 8 bits, 16 bits ... 256 bits, respectively. `uint` and `int` are aliases for `uint256` and `int256`. Similar to `uint` and `int`, `ufixed` and `fixed` represent fractional numbers. `ufixed0x8`, `ufixed0x16` ... `ufixed0x256` are used to hold unsigned fractional numbers of 8 bits, 16 bits ... 256 bits, respectively. Similarly, `fixed0x8`, `fixed0x16` ... `fixed0x256` are used to hold signed fractional numbers of 8 bits, 16 bits ... 256 bits, respectively. If it's a number requiring more than 256 bits, then 256 bits data type is used, in which case the approximation of the number is stored.

- address is used to store up to a 20 byte value by assigning a hexadecimal literal. It is used to store Ethereum addresses. The address type exposes two properties: balance and send. balance is used to check the balance of the address and send is used to transfer Ether to the address. The send method takes the amount of wei that needs to be transferred and returns true or false depending on whether the transfer was successful or not. The wei is deducted from the contract that invokes the send method. You can use the 0x prefix in Solidity to assign a hexadecimal-encoded representation of values to variables.

## Arrays

Solidity supports both generic and byte arrays. It supports both fixed size and dynamic arrays. It also supports multidimensional arrays.

bytes1, bytes2, bytes3, ..., bytes32 are types for byte arrays. byte is an alias for bytes1.

Here is an example that shows generic array syntaxes:

```
contract sample{
    //dynamic size array
    //wherever an array literal is seen a new array is created. If the
array literal is in state than it's stored in storage and if it's found
inside function than its stored in memory
    //Here myArray stores [0, 0] array. The type of [0, 0] is decided based
on its values.
    //Therefore you cannot assign an empty array literal.
    int[] myArray = [0, 0];

    function sample(uint index, int value){

        //index of an array should be uint256 type
        myArray[index] = value;

        //myArray2 holds pointer to myArray
        int[] myArray2 = myArray;

        //a fixed size array in memory
        //here we are forced to use uint24 because 99999 is the max value
and 24 bits is the max size required to hold it.
        //This restriction is applied to literals in memory because memory
is expensive. As [1, 2, 99999] is of type uint24 therefore myArray3 also
has to be the same type to store pointer to it.
        uint24[3] memory myArray3 = [1, 2, 99999]; //array literal
        //throws exception while compiling as myArray4 cannot be assigned
to complex type stored in memory
        uint8[2] myArray4 = [1, 2];
    }
}
```

Here are some important things you need to know about arrays:

- Arrays also have a `length` property that is used to find the length of an array. You can also assign a value to the length property to change the size of the array. However, you cannot resize an array in memory or resize a nondynamic array.
- If you try to access an unset index of a dynamic array, an exception is thrown.

Remember that arrays, structs, and maps cannot be parameters of functions and also cannot be returned by functions.

## Strings

In Solidity, there are two ways to create strings: using `bytes` and `string`. `bytes` is used to create a raw string, whereas `string` is used to create a UTF-8 string. The length of string is always dynamic.

Here is an example that shows string syntaxes:

```
contract sample{
    //wherever a string literal is seen a new string is created. If the
string literal is in state than it's stored in storage and if it's found
inside function than its stored in memory
    //Here myString stores "" string.
    string myString = "";  //string literal
    bytes myRawString;

    function sample(string initString, bytes rawStringInit){
        myString = initString;

        //myString2 holds a pointer to myString
        string myString2 = myString;

        //myString3 is a string in memory
        string memory myString3 = "ABCDE";

        //here the length and content changes
        myString3 = "XYZ";
        myRawString = rawStringInit;

        //incrementing the length of myRawString
        myRawString.length++;

        //throws exception while compiling
        string myString4 = "Example";

        //throws exception while compiling
        string myString5 = initString;
    }
}
```

# Structs

Solidity also supports structs. Here is an example that shows struct syntaxes:

```
contract sample{
    struct myStruct {
        bool myBool;
        string myString;
    }

    myStruct s1;

    //wherever a struct method is seen a new struct is created. If the
struct method is in state than it's stored in storage and if it's found
inside function than its stored in memory
    myStruct s2 = myStruct(true, ""); //struct method syntax

    function sample(bool initBool, string initString){

        //create a instance of struct
        s1 = myStruct(initBool, initString);

        //myStruct(initBool, initString) creates a instance in memory
        myStruct memory s3 = myStruct(initBool, initString);
    }
}
```

Note that a function parameter cannot be a struct and a function cannot return a struct.

# Enums

Solidity also supports enums. Here is an example that shows enum syntaxes:

```
contract sample {

    //The integer type which can hold all enum values and is the smallest
is chosen to hold enum values
    enum OS { Windows, Linux, OSX, UNIX }

    OS choice;

    function sample(OS chosen){
        choice = chosen;
    }

    function setLinuxOS(){
        choice = OS.Linux;
    }

    function getChoice() returns (OS chosenOS){
        return choice;
    }
}
```

# Mappings

A mapping data type is a hash table. Mappings can only live in storage, not in memory. Therefore, they are declared only as state variables. A mapping can be thought of as consisting of key/value pairs. The key is not actually stored; instead, the keccak256 hash of the key is used to look up for the value. Mappings don't have a length. Mappings cannot be assigned to another mapping.

Here is an example of how to create and use a mapping:

```
contract sample{
    mapping (int => string) myMap;

    function sample(int key, string value){
        myMap[key] = value;

        //myMap2 is a reference to myMap
        mapping (int => string) myMap2 = myMap;
    }
}
```

Remember that if you try to access an unset key, it gives us all 0 bits.

# The delete operator

The `delete` operator can be applied to any variable to reset it to its default value. The default value is all bits assigned to 0.

It we apply `delete` to a dynamic array, then it deletes all of its elements and the length becomes 0. And if we apply it to a static array, then all of its indices are reset. You can also apply `delete` to specific indices, in which case the indices are reset.

Nothing happens if you apply `delete` to a map type. But if you apply `delete` to a key of a map, then the value associated with the key is deleted.

Here is an example to demonstrate the `delete` operator:

```
contract sample {

    struct Struct {
        mapping (int => int) myMap;
        int myNumber;
    }

    int[] myArray;
    Struct myStruct;

    function sample(int key, int value, int number, int[] array) {

        //maps cannot be assigned so while constructing struct we ignore
the maps
        myStruct = Struct(number);

        //here set the map key/value
        myStruct.myMap[key] = value;

        myArray = array;
    }

    function reset(){

        //myArray length is now 0
        delete myArray;
```

```
            //myNumber is now 0 and myMap remains as it is
            delete myStruct;
        }

        function deleteKey(int key){

            //here we are deleting the key
            delete myStruct.myMap[key];
        }

    }
```

## Conversion between elementary types

Other than arrays, strings, structs, enums, and maps, everything else is called elementary types.

If an operator is applied to different types, the compiler tries to implicitly convert one of the operands into the type of the other. In general, an implicit conversion between value-types is possible if it makes sense semantically and no information is lost: `uint8` is convertible to `uint16` and `int128` to `int256`, but `int8` is not convertible to `uint256` (because `uint256` cannot hold, for example, -1). Furthermore, unsigned integers can be converted into bytes of the same or larger size, but not vice versa. Any type that can be converted into `uint160` can also be converted into `address`.

Solidity also supports explicit conversion. So if the compiler doesn't allow implicit conversion between two data types, then you can go for explicit conversion. It is always recommended that you avoid explicit conversion because it may give you unexpected results.

Let's look at an example of explicit conversion:

```
    uint32 a = 0x12345678;
    uint16 b = uint16(a); // b will be 0x5678 now
```

Here we are converting `uint32` type to `uint16` explicitly, that is, converting a large type to a smaller type; therefore, higher-order bits are cut-off.

# Using var

Solidity provides the `var` keyword to declare variables. The type of the variable in this case is decided dynamically depending on the first value assigned to it. Once a value is assigned, the type is fixed, so if you assign another type to it, it will cause type conversion.

Here is an example to demonstrate `var`:

```
int256 x = 12;

//y type is int256
var y = x;

uint256 z= 9;

//exception because implicit conversion not possible
y = z;
```

Remember that `var` cannot be used when defining arrays and maps. And it cannot be used to define function parameters and state variables.

# Control structures

Solidity supports `if`, `else`, `while`, `for`, `break`, `continue`, `return`, `? :` control structures.

Here is an example to demonstrate the control structures:

```
contract sample{
    int a = 12;
    int[] b;

    function sample()
    {
        //"==" throws exception for complex types
        if(a == 12)
        {
        }
        else if(a == 34)
        {
        }
        else
        {
        }
```

```
            var temp = 10;

            while(temp < 20)
            {
                if(temp == 17)
                {
                    break;
                }
                else
                {
                    continue;
                }

                temp++;
            }

            for(var iii = 0; iii < b.length; iii++)
            {

            }
        }
    }
```

# Creating contracts using the new operator

A contract can create a new contract using the `new` keyword. The complete code of the contract being created has to be known.

Here is an example to demonstrate this:

```
    contract sample1
    {
        int a;

        function assign(int b)
        {
            a = b;
        }
    }


    contract sample2{
        function sample2()
        {
            sample1 s = new sample1();
            s.assign(12);
        }
    }
```

# Exceptions

There are some cases where exceptions are thrown automatically. You can use `throw` to throw an exception manually. The effect of an exception is that the currently executing call is stopped and reverted (that is, all changes to the state and balances are undone). Catching exceptions is not possible:

```
    contract sample
    {
        function myFunction()
        {
            throw;
        }
    }
```

# External function calls

There are two kinds of function calls in Solidity: internal and external function calls. An internal function call is when a function calls another function in the same contract.

An external function call is when a function calls a function of another contract. Let's look at an example:

```
    contract sample1
    {
        int a;

        //"payable" is a built-in modifier
        //This modifier is required if another contract is sending Ether while
    calling the method
        function sample1(int b) payable
        {
            a = b;
        }

        function assign(int c)
        {
            a = c;
        }
        function makePayment(int d) payable
        {
            a = d;
        }
    }


    contract sample2{

        function hello()
        {
        }

        function sample2(address addressOfContract)
        {
            //send 12 wei while creating contract instance
            sample1 s = (new sample1).value(12)(23);

            s.makePayment(22);

            //sending Ether also
            s.makePayment.value(45)(12);

            //specifying the amount of gas to use
            s.makePayment.gas(895)(12);

            //sending Ether and also specifying gas
            s.makePayment.value(4).gas(900)(12);

            //hello() is internal call whereas this.hello() is external call
            this.hello();

            //pointing a contract that's already deployed
            sample1 s2 = sample1(addressOfContract);

            s2.makePayment(112);

        }
    }
```

Calls made using the `this` keyword are called as external calls. The `this` keyword inside functions represents the current contract instance.

# Features of contracts

Now it's time to get deeper into contracts. We will look at some new features and also get deeper into the features we have already seen.

## Visibility

The visibility of a state variable or a function defines who can see it. There are four kinds of visibilities for function and state variables: `external`, `public`, `internal`, and `private`.

By default, the visibility of functions is `public` and the visibility of state variables is `internal`. Let's look at what each of these visibility functions mean:

- `external`: External functions can be called only from other contracts or via transactions. An external function `f` cannot be called internally; that is, `f()` will not work, but `this.f()` works. You cannot apply the `external` visibility to state variables.
- `public`: Public functions and state variables can be accessed in all ways possible. The compiler generated accessor functions are all public state variables. You cannot create your own accessors. Actually, it generates only getters, not setters.
- `internal`: Internal functions and state variables can only be accessed internally, that is, from within the current contract and the contracts inheriting it. You cannot use `this` to access it.
- `private`: Private functions and state variables are like internal ones, but they cannot be accessed by the inheriting contracts.

Here is a code example to demonstrate visibility and accessors:

```
contract sample1
{
    int public b = 78;
    int internal c = 90;

    function sample1()
    {
        //external access
        this.a();

        //compiler error
        a();

        //internal access
        b = 21;

        //external access
        this.b;

        //external access
        this.b();

        //compiler error
        this.b(8);

        //compiler error
        this.c();

        //internal access
        c = 9;
    }

    function a() external
    {

    }
}

contract sample2
{
    int internal d = 9;
    int private e = 90;
}

//sample3 inherits sample2
contract sample3 is sample2
{
    sample1 s;

    function sample3()
    {
        s = new sample1();

        //external access
        s.a();

        //external access
        var f = s.b;

        //compiler error as accessor cannot used to assign a value
        s.b = 18;
        //compiler error
        s.c();

        //internal access
        d = 8;

        //compiler error
        e = 7;
    }
}
```

# Function modifiers

We saw earlier what a function modifier is, and we wrote a basic function modifier. Now let's look at modifiers in depth.

Modifiers are inherited by child contracts, and child contracts can override them. Multiple modifiers can be applied to a function by specifying them in a whitespace-separated list and will be evaluated in order. You can also pass arguments to modifiers.

Inside the modifier, the next modifier body or function body, whichever comes next, is inserted where `_;` appears.

Let's take a look at a complex code example of function modifiers:

```
contract sample
{
    int a = 90;

    modifier myModifier1(int b) {
        int c = b;
        _;
        c = a;
        a = 8;
    }

    modifier myModifier2 {
        int c = a;
        _;
    }

    modifier myModifier3 {
        a = 96;
        return;
        _;
        a = 99;
    }

    modifier myModifier4 {
        int c = a;
        _;
    }

    function myFunction() myModifier1(a) myModifier2 myModifier3 returns
(int d)
    {
        a = 1;
        return a;
    }
}
```

This is how `myFunction()` is executed:

```
int c = b;
    int c = a;
        a = 96;
        return;
            int c = a;
                a = 1;
                return a;
        a = 99;
c = a;
a = 8;
```

Here, when you call the `myFunction` method, it will return `0`. But after that, when you try to access the state variable `a`, you will get `8`.

`return` in a modifier or function body immediately leaves the whole function and the return value is assigned to whatever variable it needs to be.

In the case of functions, the code after `return` is executed after the caller's code execution is finished. And in the case of modifiers, the code after `_;` in the previous modifier is executed after the caller's code execution is finished. In the earlier example, line numbers 5, 6, and 7 are never executed. After line number 4, the execution starts from line numbers 8 to 10.

`return` inside modifiers cannot have a value associated with it. It always returns 0 bits.

# The fallback function

A contract can have exactly one unnamed function called the `fallback` function. This function cannot have arguments and cannot return anything. It is executed on a call to the contract if none of the other functions match the given function identifier.

This function is also executed whenever the contract receives Ether without any function call; that is, the transaction sends Ether to the contracts and doesn't invoke any method. In such a context, there is usually very little gas available to the function call (to be precise, 2,300 gas), so it is important to make fallback functions as cheap as possible.

Contracts that receive Ether but do not define a fallback function throw an exception, sending back the Ether. So if you want your contract to receive Ether, you have to implement a fallback function.

Here is an example of a fallback function:

```
contract sample
{
    function() payable
    {
        //keep a note of how much Ether has been sent by whom
    }
}
```

# Inheritance

Solidity supports multiple inheritance by copying code including polymorphism. Even if a contract inherits from multiple other contracts, only a single contract is created on the blockchain; the code from the parent contracts is always copied into the final contract.

Here is an example to demonstrate inheritance:

```
contract sample1
{
    function a(){}

    function b(){}
}


//sample2 inherits sample1
contract sample2 is sample1
{
    function b(){}
}

contract sample3
{
    function sample3(int b)
    {

    }
}

//sample4 inherits from sample1 and sample2
//Note that sample1 is also parent of sample2, yet there is only a single
instance of sample1
contract sample4 is sample1, sample2
{
    function a(){}

    function c(){

        //this executes the "a" method of sample3 contract
        a();

        //this executes the 'a" method of sample1 contract
        sample1.a();

        //calls sample2.b() because it's in last in the parent contracts
list and therefore it overrides sample1.b()
        b();
    }
}

//If a constructor takes an argument, it needs to be provided at the
constructor of the child contract.
//In Solidity child constructor doesn't call parent constructor instead
parent is initialized and copied to child
contract sample5 is sample3(122)
{

}
```

## The super keyword

The `super` keyword is used to refer to the next contract in the final inheritance chain. Let's take a look at an example to understand this:

```
contract sample1
{
}

contract sample2
{
}

contract sample3 is sample2
{
}

contract sample4 is sample2
{
}

contract sample5 is sample4
{
    function myFunc()
    {
    }
}

contract sample6 is sample1, sample2, sample3, sample5
{
    function myFunc()
    {
        //sample5.myFunc()
        super.myFunc();
    }
}
```

The final inheritance chain with respect to the `sample6` contract is `sample6`, `sample5`, `sample4`, `sample2`, `sample3`, `sample1`. The inheritance chain starts with the most derived contracts and ends with the least derived contract.

## Abstract contracts

Contracts that only contain the prototype of functions instead of implementation are called abstract contracts. Such contracts cannot be compiled (even if they contain implemented functions alongside nonimplemented functions). If a contract inherits from an abstract contract and does not implement all nonimplemented functions by overriding, it will itself be abstract.

These abstract contracts are only provided to make the interface known to the compiler. This is useful when you are referring to a deployed contract and calling its functions.

Here is an example to demonstrate this:

```
contract sample1
{
    function a() returns (int b);
}


contract sample2
{
    function myFunc()
    {
        sample1 s = sample1(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);

        //without abstract contract this wouldn't have compiled
        s.a();
    }
}
```

# Libraries

Libraries are similar to contracts, but their purpose is that they are deployed only once at a specific address and their code is reused by various contracts. This means that if library functions are called, their code is executed in the context of the calling contract; that is, `this` points to the calling contract, and especially, the storage from the calling contract can be accessed. As a library is an isolated piece of source code, it can only access state variables of the calling contract if they are explicitly supplied (it would have no way to name them otherwise).

Libraries cannot have state variables; they don't support inheritance and they cannot receive Ether. Libraries can contain structs and enums.

Once a Solidity library is deployed to the blockchain, it can be used by anyone, assuming you know its address and have the source code (with only prototypes or complete implementation). The source code is required by the Solidity compiler so that it can make sure that the methods you are trying to access actually exist in the library.

Let's take a look at an example:

```
library math
{
    function addInt(int a, int b) returns (int c)
    {
        return a + b;
    }
}

contract sample
{
    function data() returns (int d)
    {
        return math.addInt(1, 2);
    }
}
```

We cannot add the address of the library in the contract source code; instead, we need to provide the library address during compilation to the compiler.

Libraries have many use cases. The two major use cases of libraries are as follows:

- If you have many contracts that have some common code, then you can deploy that common code as a library. This will save gas as gas depends on the size of the contract too. Therefore, we can think of a library as a base contract of the contract that uses it. Using a base contract instead of a library to split the common code won't save gas because in Solidity, inheritance works by copying code. Due to the reason that libraries are thought of as base contracts, functions with the internal visibility in a library are copied to the contract that uses it; otherwise, functions with the internal visibility of a library cannot be called by the contract that uses the library, as an external call would be required and functions with the internal visibility cannot be invoked using the external call. Also, structs and enums in a library are copied to the contract that uses the library.

- Libraries can be used to add member functions to data types.

If a library contains only internal functions and/or structs/enums, then the library doesn't need to be deployed, as everything that's there in the library is copied to the contract that uses it.

# Using for

The `using A for B;` directive can be used to attach library functions (from the library `A` to any type `B`). These functions will receive the object they are called on as their first parameter.

The effect of `using A for *;` is that the functions from the library `A` are attached to all types.

Here is an example to demonstrate `for`:

```
library math
{
    struct myStruct1 {
        int a;
    }

    struct myStruct2 {
        int a;
    }

    //Here we have to make 's' location storage so that we get a reference.
    //Otherwise addInt will end up accessing/modifying a different instance
of myStruct1 than the one on which its invoked
    function addInt(myStruct1 storage s, int b) returns (int c)
    {
        return s.a + b;
    }

    function subInt(myStruct2 storage s, int b) returns (int c)
    {
        return s.a + b;
    }
}

contract sample
{
    //"*" attaches the functions to all the structs
    using math for *;
    math.myStruct1 s1;
    math.myStruct2 s2;

    function sample()
    {
        s1 = math.myStruct1(9);
        s2 = math.myStruct2(9);

        s1.addInt(2);

        //compiler error as the first parameter of addInt is of type
myStruct1 so addInt is not attached to myStruct2
        s2.addInt(1);
    }
}
```

# Returning multiple values

Solidity allows functions to return multiple values. Here is an example to demonstrate this:

```
contract sample
{
    function a() returns (int a, string c)
    {
        return (1, "ss");
    }

    function b()
    {
        int A;
        string memory B;

        //A is 1 and B is "ss"
        (A, B) = a();

        //A is 1
        (A,) = a();

        //B is "ss"
        (, B) = a();
    }
}
```

# Importing other Solidity source files

Solidity allows a source file to import other source files. Here is an example to demonstrate this:

```
//This statement imports all global symbols from "filename" (and symbols
imported there) into the current global scope. "filename" can be a absolute
or relative path. It can only be a HTTP URL
import "filename";

//creates a new global symbol symbolName whose members are all the global
symbols from "filename".
import * as symbolName from "filename";

//creates new global symbols alias and symbol2 which reference symbol1 and
symbol2 from "filename", respectively.
import {symbol1 as alias, symbol2} from "filename";

//this is equivalent to import * as symbolName from "filename";.
import "filename" as symbolName;
```

# Globally available variables

There are special variables and functions that always exist globally. They are discussed in the upcoming sections.

## Block and transaction properties

The block and transaction properties are as follows:

- `block.blockhash(uint blockNumber) returns (bytes32)`: The hash of the given block only works for the 256 most recent blocks.
- `block.coinbase (address)`: The current block miner's address.
- `block.difficulty (uint)`: The current block difficulty.
- `block.gaslimit (uint)`: The current block gas limit. It defines the maximum amount of gas that all transactions in the whole block combined are allowed to consume. Its purpose is to keep the block propagation and processing time low, thereby allowing a sufficiently decentralized network. Miners have the right to set the gas limit for the current block to be within ~0.0975% (1/1,024) of the gas limit of the last block, so the resulting gas limit should be the median of miners' preferences.

- `block.number (uint)`: The current block number.
- `block.timestamp (uint)`: The current block timestamp.
- `msg.data (bytes)`: The complete call data holds the function and its arguments that the transaction invokes.
- `msg.gas (uint)`: The remaining gas.
- `msg.sender (address)`: The sender of the message (the current call).
- `msg.sig (bytes4)`: The first four bytes of the call data (the function identifier).
- `msg.value (uint)`: The number of wei sent with the message.
- `now (uint)`: The current block timestamp (alias for `block.timestamp`).
- `tx.gasprice (uint)`: The gas price of the transaction.
- `tx.origin (address)`: The sender of the transaction (full call chain).

## Address type related

The address type related variables are as follows:

- `<address>.balance (uint256)`: The balance of the address in wei
- `<address>.send(uint256 amount) returns (bool)`: Sends the given amount of wei to `address`; returns `false` on failure

## Contract related

The contract related variables are as follows:

- `this`: The current contract, explicitly convertible to the `address` type.
- `selfdestruct(address recipient)`: Destroys the current contract, sending its funds to the given address.

# Ether units

A literal number can take a suffix of `wei`, `finney`, `szabo`, or `Ether` to convert between the subdenominations of Ether, where Ether currency numbers without a postfix are assumed to be wei; for example, `2 Ether == 2000 finney` evaluates to `true`.

# Proof of existence, integrity, and ownership contract

Let's write a Solidity contract that can prove file ownership without revealing the actual file. It can prove that the file existed at a particular time and finally check for document integrity.

We will achieve proof of ownership by storing the hash of the file and the owner's name as pairs. We will achieve proof of existence by storing the hash of the file and the block timestamp as pairs. Finally, storing the hash itself proves the file integrity; that is, if the file was modified, then its hash will change and the contract won't be able to find any such file, therefore proving that the file was modified.

Here is the code for the smart contract to achieve all this:

```
contract Proof
{
    struct FileDetails
    {
        uint timestamp;
        string owner;
    }

    mapping (string => FileDetails) files;

    event logFileAddedStatus(bool status, uint timestamp, string owner,
string fileHash);

    //this is used to store the owner of file at the block timestamp
    function set(string owner, string fileHash)
    {
        //There is no proper way to check if a key already exists or not
therefore we are checking for default value i.e., all bits are 0
        if(files[fileHash].timestamp == 0)
        {
            files[fileHash] = FileDetails(block.timestamp, owner);

            //we are triggering an event so that the frontend of our app
knows that the file's existence and ownership details have been stored
            logFileAddedStatus(true, block.timestamp, owner, fileHash);
        }
        else
        {
            //this tells to the frontend that file's existence and
ownership details couldn't be stored because the file's details had already
been stored earlier
            logFileAddedStatus(false, block.timestamp, owner, fileHash);
        }
    }

    //this is used to get file information
    function get(string fileHash) returns (uint timestamp, string owner)
    {
        return (files[fileHash].timestamp, files[fileHash].owner);
    }
}
```

# Compiling and deploying contracts

Ethereum provides the solc compiler, which provides a command-line interface to compile `.sol` files. Visit `http://solidity.readthedocs.io/en/develop/installing-solidity.html#binary-packages` to find instructions to install it and visit `https://Solidity.readthedocs.io/en/develop/using-the-compiler.html` to find instructions on how to use it. We won't be using the solc compiler directly; instead, we will be using solcjs and Solidity browser. Solcjs allows us to compile Solidity programmatically in Node.js, whereas browser Solidity is an IDE, which is suitable for small contracts.

For now, let's just compile the preceding contract using a browser Solidity provided by Ethereum. Learn more about it at `https://Ethereum.github.io/browser-Solidity/`. You can also download this browser Solidity source code and use it offline. Visit `https://github.com/Ethereum/browser-Solidity/tree/gh-pages` to download it.

A major advantage of using this browser Solidity is that it provides an editor and also generates code to deploy the contract.

In the editor, copy and paste the preceding contract code. You will see that it compiles and gives you the web3.js code to deploy it using the geth interactive console.

You will get this output:

```
var proofContract =
web3.eth.contract([{"constant":false,"inputs":[{"name":"fileHash","type":"s
tring"}],"name":"get","outputs":[{"name":"timestamp","type":"uint256"},{"na
me":"owner","type":"string"}],"payable":false,"type":"function"},{"constant
":false,"inputs":[{"name":"owner","type":"string"},{"name":"fileHash","type
":"string"}],"name":"set","outputs":[],"payable":false,"type":"function"},{
"anonymous":false,"inputs":[{"indexed":false,"name":"status","type":"bool"}
,{"indexed":false,"name":"timestamp","type":"uint256"},{"indexed":false,"na
me":"owner","type":"string"},{"indexed":false,"name":"fileHash","type":"str
ing"}],"name":"logFileAddedStatus","type":"event"}]);
var proof = proofContract.new(
  {
    from: web3.eth.accounts[0],
    data: '60606040526......,
    gas: 4700000
  }, function (e, contract){
   console.log(e, contract);
  if (typeof contract.address !== 'undefined') {
    console.log('Contract mined! address: ' + contract.address + '
transactionHash: ' + contract.transactionHash);
  }
})
```

`data` represents the compiled version of the contract (bytecode) that the EVM understands. The source code is first converted into opcode, and then opcode are converted into bytecode. Each opcode has gas associated with it.

The first argument to the `web3.eth.contract` is the ABI definition. The ABI definition is used when creating transactions, as it contains the prototype of all the methods.

Now run geth in the developer mode with the mining enabled. To do this, run the following command:

```
geth --dev --mine
```

Now open another command-line window and in that, enter this command to open geth's interactive JavaScript console:

```
geth attach
```

This should connect the JS console to the geth instance running in the other window.

On the right-hand side panel of the browser Solidity, copy everything that's there in the web3 deploy textarea and paste it in the interactive console. Now press *Enter*. You will first get the transaction hash, and after waiting for some time, you will get the contract address after the transaction is mined. The transaction hash is the hash of the transaction, which is unique for every transaction. Every deployed contract has a unique contract address to identity the contract in the blockchain.

The contract address is deterministically computed from the address of its creator (the from address) and the number of transactions the creator has sent (the transaction nonce). These two are RLP-encoded and then hashed using the keccak-256 hashing algorithm. We will learn more about the transaction nonce later. You can learn more about RLP at `https://github.com/Ethereum/wiki/wiki/RLP`.

Now let's store the file details and retrieve them.

Place this code to broadcast a transaction to store a file's details:

```
var contract_obj =
proofContract.at("0x9220c8ec6489a4298b06c2183cf04fb7e8fbd6d4");
contract_obj.set.sendTransaction("Owner Name",
"e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855", {
 from: web3.eth.accounts[0],
}, function(error, transactionHash){
  if (!err)
    console.log(transactionHash);
})
```

Here, replace the contract address with the contract address you got. The first argument of the `proofContract.at` method is the contract address. Here, we didn't provide the gas, in which case, it's automatically calculated.

Now let's find the file's details. Run this code in order to find the file's details:

```
contract_obj.get.call("e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495
991b7852b855");
```

You will get this output:

```
[1477591434, "Owner Name"]
```

The call method is used to call a contract's method on EVM with the current state. It doesn't broadcast a transaction. To read data, we don't need to broadcast because we will have our own copy of the blockchain.

We will learn more about web3.js in the coming chapters.

# Summary

In this chapter, we learned the Solidity programming language. We learned about data location, data types, and advanced features of contracts. We also learned the quickest and easiest way to compile and deploy a smart contract. Now you should be comfortable with writing smart contracts.

In the next chapter, we will build a frontend for the smart contract, which will make it easy to deploy the smart contract and run transactions.

# 4
# Getting Started with web3.js

In the previous chapter, we learned how to write smart contracts and used geth's interactive console to deploy and broadcast transactions using web3.js. In this chapter, we will learn web3.js and how to import, connect to geth, and use it in Node.js or client-side JavaScript. We will also learn how to build a web client using web3.js for the smart contract that we created in the previous chapter.

In this chapter, we'll cover the following topics:

- Importing web3.js in Node.js and client-side JavaScript
- Connecting to geth
- Exploring the various things that can be done using web3.js
- Discovering various most used APIs of web3.js
- Building a Node.js application for an ownership contract

## Introduction to web3.js

web3.js provides us with JavaScript APIs to communicate with geth. It uses JSON-RPC internally to communicate with geth. web3.js can also communicate with any other kind of Ethereum node that supports JSON-RPC. It exposes all JSON-RPC APIs as JavaScript APIs; that is, it doesn't just support all the Ethereum-related APIs; it also supports APIs related to Whisper and Swarm.

You will learn more and more about web3.js as we build various projects, but for now, let's go through some of the most used APIs of web3.js and then we will build a frontend for our ownership smart contract using web3.js.

At the time of writing this, the latest version of web3.js is 0.16.0. We will learn everything with respect to this version.

web3.js is hosted at `https://github.com/ethereum/web3.js` and the complete documentation is hosted at `https://github.com/ethereum/wiki/wiki/JavaScript-API`.

## Importing web3.js

To use web3.js in Node.js, you can simply run `npm install web3` inside your project directory, and in the source code, you can import it using `require("web3");`.

To use web3.js in client-side JavaScript, you can enqueue the `web3.js` file, which can be found inside the `dist` directory of the project source code. Now you will have the `Web3` object available globally.

## Connecting to nodes

web3.js can communicate with nodes using HTTP or IPC. We will use HTTP to set up communication with nodes. web3.js allows us to establish connections with multiple nodes. An instance of `web3` represents a connection with a node. The instance exposes the APIs.

When an app is running inside Mist, it automatically makes an instance of `web3` available that's connected to the mist node. The variable name of the instance is `web3`.

Here is the basic code to connect to a node:

```
if (typeof web3 !== 'undefined') {
  web3 = new Web3(new
Web3.providers.HttpProvider("http://localhost:8545"));
}
```

At first, we check here whether the code is running inside mist by checking whether `web3` is `undefined` or not. If `web3` is defined, then we use the already available instance; otherwise, we create an instance by connecting to our custom node. If you want to connect to the custom node regardless of whether the app is running inside mist or not, then remove the `if` condition form the preceding code. Here, we are assuming that our custom node is running locally on port number `8545`.

The `Web3.providers` object exposes constructors (called providers in this context) to establish connection and transfer messages using various protocols. `Web3.providers.HttpProvider` lets us establish an HTTP connection, whereas `Web3.providers.IpcProvider` lets us establish an IPC connection.

The web3.currentProvider property is automatically assigned to the current provider instance. After creating a web3 instance, you can change its provider using the web3.setProvider() method. It takes one argument, that is, the instance of the new provider.

Remember that geth has HTTP-RPC disabled by default. So enable it by passing the --rpc option while running geth. By default, HTTP-RPC runs on port 8545.

web3 exposes a isConnected() method, which can be used to check whether it's connected to the node or not. It returns true or false depending on the connection status.

# The API structure

web3 contains an eth object (web3.eth) specifically for Ethereum blockchain interactions and an shh object (web3.shh) for whisper interaction. Most APIs of web3.js are inside these two objects.

All the APIs are synchronous by default. If you want to make an asynchronous request, you can pass an optional callback as the last parameter to most functions. All callbacks use an error-first callback style.

Some APIs have an alias for asynchronous requests. For example, web3.eth.coinbase() is synchronous, whereas web3.eth.getCoinbase() is asynchronous.

Here is an example:

```
//sync request
try
{
  console.log(web3.eth.getBlock(48));
}
catch(e)
{
  console.log(e);
}

//async request
web3.eth.getBlock(48, function(error, result){
    if(!error)
        console.log(result)
    else
```

```
        console.error(error);
})
```

getBlock is used to get information on a block using its number or hash. Or, it can take a string such as "earliest" (the genesis block), "latest" (the top block of the blockchain), or "pending" (the block that's being mined). If you don't pass an argument, then the default is web3.eth.defaultBlock, which is assigned to "latest" by default.

All the APIs that need a block identification as input can take a number, hash, or one of the readable strings. These APIs use web3.eth.defaultBlock by default if the value is not passed.

# BigNumber.js

JavaScript is natively poor at handling big numbers correctly. Therefore, applications that require you to deal with big numbers and need perfect calculations use the BigNumber.js library to work with big numbers.

web3.js also depends on BigNumber.js. It adds it automatically. web3.js always returns the BigNumber object for number values. It can take JavaScript numbers, number strings, and BigNumber instances as input.

Here is an example to demonstrate this:

```
web3.eth.getBalance("0x27E829fB34d14f3384646F938165dfcD30cFfB7c").toString(
);
```

Here, we use the web3.eth.getBalance() method to get the balance of an address. This method returns a BigNumber object. We need to call toString() on a BigNumber object to convert it into a number string.

BigNumber.js fails to correctly handle numbers with more than 20 floating point digits; therefore, it is recommended that you store the balance in a wei unit and while displaying, convert it to other units. web3.js itself always returns and takes the balance in wei. For example, the getBalance() method returns the balance of the address in the wei unit.

# Unit conversion

web3.js provides APIs to convert the wei balance into any other unit and any other unit balance into wei.

The `web3.fromWei()` method is used to convert a wei number into any other unit, whereas the `web3.toWei()` method is used to convert a number in any other unit into wei. Here is example to demonstrate this:

```
web3.fromWei("1000000000000000000", "ether");
web3.toWei("0.000000000000000001", "ether");
```

In the first line, we convert wei into ether, and in the second line, we convert ether into wei. The second argument in both methods can be one of these strings:

- `kwei/ada`
- `mwei/babbage`
- `gwei/shannon`
- `szabo`
- `finney`
- `ether`
- `kether/grand/einstein`
- `mether`
- `gether`
- `tether`

# Retrieving gas price, balance, and transaction details

Let's take a look at the APIs to retrieve the gas price, the balance of an address, and information on a mined transaction:

```
//It's sync. For async use getGasPrice
console.log(web3.eth.gasPrice.toString());

console.log(web3.eth.getBalance("0x407d73d8a49eeb85d32cf465507dd71d507100c1
", 45).toString());

console.log(web3.eth.getTransactionReceipt("0x9fc76417374aa880d4449a1f7f31e
c597f00b1f6f3dd2d66f4c9c6c445836d8b"));
```

The output will be of this form:

```
20000000000
30000000000
{
  "transactionHash":
"0x9fc76417374aa880d4449a1f7f31ec597f00b1f6f3dd2d66f4c9c6c445836d8b ",
  "transactionIndex": 0,
  "blockHash":
"0xef95f2f1ed3ca60b048b4bf67cde2195961e0bba6f70bcbea9a2c4e133e34b46",
  "blockNumber": 3,
  "contractAddress": "0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b",
  "cumulativeGasUsed": 314159,
  "gasUsed": 30234
}
```

Here is how the preceding method works:

- `web3.eth.gasPrice()`: Determines the gas price by the x latest blocks' median gas price.
- `web3.ethgetBalance()`: Returns the balance of any given address. All the hashes should be provided as hexadecimal strings to the web3.js APIs, not as hexadecimal literals. The input for the solidity `address` type should also be hexadecimal strings.
- `web3.eth.getTransactionReceipt()`: This is used to get details about a transaction using its hash. It returns a transaction receipt object if the transaction was found in the blockchain; otherwise, it returns null. The transaction receipt object contains the following properties:
    - `blockHash`: The hash of the block where this transaction was
    - `blockNumber`: The block number where this transaction was
    - `transactionHash`: The hash of the transaction
    - `transactionIndex`: The integer of the transactions' index position in the block
    - `from`: The address of the sender
    - `to`: The address of the receiver; `null` when it's a contract creation transaction
    - `cumulativeGasUsed`: The total amount of gas used when this transaction was executed in the block
    - `gasUsed`: The amount of gas used by this specific transaction alone

- `contractAddress`: The contract address created if the transaction was a contract creation; otherwise, null
- `logs`: The array of log objects that this transaction generated

## Sending ether

Let's look at how to send ether to any address. To send ether, you need to use the `web3.eth.sendTransaction()` method. This method can be used to send any kind of transaction but is mostly used to send ether because deploying a contract or calling a method of contract using this method is cumbersome as it requires you to generate the data of the transaction rather than automatically generating it. It takes a transaction object that has the following properties:

- `from`: The address for the sending account. Uses the `web3.eth.defaultAccount` property if not specified.
- `to`: This is optional. It's the destination address of the message and is left undefined for a contract-creation transaction.
- `value`: This is optional. The value is transferred for the transaction in wei as well as the endowment if it's a contract-creation transaction.
- `gas`: This is optional. It's the amount of gas to use for the transaction (unused gas is refunded). If not provided, then it's automatically determined.
- `gasPrice`: This is optional. It's the price of gas for this transaction in wei, and it defaults to the mean network gas price.
- `data`: This is optional. It's either a byte string containing the associated data of the message, or in the case of a contract-creation transaction, the initialization code.
- `nonce`: This is optional. It's an integer. Every transaction has a nonce associated with it. A nonce is a counter that indicates the number of transactions sent by the sender of the transaction. If not provided, then it is automatically determined. It helps prevent replay attacks. This nonce is not the nonce associated with a block. If we are using a nonce greater than the nonce the transaction should have, then the transaction is put in a queue until the other transactions arrive. For example, if the next transaction nonce should be 4 and if we set the nonce to 10, then geth will wait for the middle six transactions before broadcasting this transaction. The transaction with nonce 10 is called a queued transaction, and it's not a pending transaction.

Let's look at an example of how to send ether to an address:

```
var txnHash = web3.eth.sendTransaction({
  from: web3.eth.accounts[0],
  to: web3.eth.accounts[1],
  value: web3.toWei("1", "ether")
});
```

Here, we send one ether from account number 0 to account number 1. Make sure that both the accounts are unlocked using the `unlock` option while running geth. In the geth interactive console, it prompts for passwords, but the web3.js API outside of the interactive console will throw an error if the account is locked. This method returns the transaction hash of the transaction. You can then check whether the transaction is mined or not using the `getTransactionReceipt()` method.

You can also use the `web3.personal.listAccounts()`, `web3.personal.unlockAccount(addr, pwd)`, and `web3.personal.newAccount(pwd)` APIs to manage accounts at runtime.

# Working with contracts

Let's learn how to deploy a new contract, get a reference to a deployed contract using it's address, send ether to a contract, send a transaction to invoke a contract method, and estimate the gas of a method call.

To deploy a new contract or to get a reference to an already deployed contract, you need to first create a contract object using the `web3.eth.contract()` method. It takes the contract ABI as an argument and returns the contract object.

Here is the code to create a contract object:

```
var proofContract =
web3.eth.contract([{"constant":false,"inputs":[{"name":"fileHash","type":"s
tring"}],"name":"get","outputs":[{"name":"timestamp","type":"uint256"},{"na
me":"owner","type":"string"}],"payable":false,"type":"function"},{"constant
":false,"inputs":[{"name":"owner","type":"string"},{"name":"fileHash","type
":"string"}],"name":"set","outputs":[],"payable":false,"type":"function"},{
"anonymous":false,"inputs":[{"indexed":false,"name":"status","type":"bool"}
,{"indexed":false,"name":"timestamp","type":"uint256"},{"indexed":false,"na
me":"owner","type":"string"},{"indexed":false,"name":"fileHash","type":"str
ing"}],"name":"logFileAddedStatus","type":"event"}]);
```

Once you have the contract, you can deploy it using the `new` method of the contract object or get a reference to an already deployed contract that matches the ABI using the `at` method.

Let's take a look at an example of how to deploy a new contract:

```
var proof = proofContract.new({
     from: web3.eth.accounts[0],
     data: "0x606060405261068...",
     gas: "4700000"
    },
    function (e, contract){
    if(e)
    {
    console.log("Error " + e);
}
else if(contract.address != undefined)
  {
    console.log("Contract Address: " + contract.address);
   }
else
  {
    console.log("Txn Hash: " + contract.transactionHash)
  }
})
```

Here, the `new` method is called asynchronously, so the callback is fired twice if the transaction was created and broadcasted successfully. The first time, it's called after the transaction is broadcasted, and the second time, it's called after the transaction is mined. If you don't provide a callback, then the `proof` variable will have the `address` property set to `undefined`. Once the contract is mined, the `address` property will be set.

In the `proof` contract, there is no constructor, but if there is a constructor, then the arguments for the constructor should be placed at the beginning of the `new` method. The object we passed contains the from address, the byte code of the contract, and the maximum gas to use. These three properties must be present; otherwise, the transaction won't be created. This object can have the properties that are present in the object passed to the `sendTransaction()` method, but here, `data` is the contract byte code and the `to` property is ignored.

You can use the `at` method to get a reference to an already deployed contract. Here is the code to demonstrate this:

```
var proof = proofContract.at("0xd45e541ca2622386cd820d1d3be74a86531c14a1");
```

Now let's look at how to send a transaction to invoke a method of a contract. Here is an example to demonstrate this:

```
proof.set.sendTransaction("Owner Name",
"e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855", {

from: web3.eth.accounts[0],
}, function(error, transactionHash){

if (!err)

console.log(transactionHash);
})
```

Here, we call the `sendTransaction` method of the object of the method namesake. The object passed to this `sendTransaction` method has the same properties as `web3.eth.sendTransaction()`, except that the `data` and `to` properties are ignored.

If you want to invoke a method on the node itself instead of creating a transaction and broadcasting it, then you can use `call` instead of `sendTransaction`. Here is an example to demonstrate this:

```
var returnValue =
proof.get.call("e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b785
2b855");
```

Sometimes, it is necessary to find out the gas that would be required to invoke a method so that you can decide whether to invoke it or not. `web3.eth.estimateGas` can be used for this purpose. However, using `web3.eth.estimateGas()` directly requires you to generate the data of the transaction; therefore, we can use the `estimateGas()` method of the object of the method namesake. Here is an example to demonstrate this:

```
var estimatedGas =
proof.get.estimateGas("e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495
991b7852b855");
```

If you want to just send some ether to a contract without invoking any method, then you can simply use the `web3.eth.sendTransaction` method.

## Retrieving and listening to contract events

Now let's look at how to watch for events from a contract. Watching for events is very important because the result of method invocations by transactions are usually returned by triggering events.

Before we get into how to retrieve and watch for events, we need to learn indexed parameters of events. A maximum of three parameters of an event can have the `indexed` attribute. This attribute is used to signal the node to index it so that the app client can search for events with matching return values. If you don't use the indexed attribute, then it will have to retrieve all the events from the node and filter the ones needed. For example, you can write the `logFileAddedStatus` event this way:

```
event logFileAddedStatus(bool indexed status, uint indexed timestamp,
string owner, string indexed fileHash);
```

Here is an example to demonstrate how to listen to contract events:

```
var event = proof.logFileAddedStatus(null, {
fromBlock: 0,
toBlock: "latest"
});
event.get(function(error, result){
if(!error)
{
  console.log(result);
}
else
{
  console.log(error);
}
})
event.watch(function(error, result){
if(!error)
{
  console.log(result.args.status);
}
else
{
  console.log(error);
}
})
setTimeout(function(){
event.stopWatching();
}, 60000)
```

```
 var events = proof.allEvents({
fromBlock: 0,
 toBlock: "latest"
});
events.get(function(error, result){
if(!error)
{
  console.log(result);
}
else
{
  console.log(error);
}
})
events.watch(function(error, result){
if(!error)
{
  console.log(result.args.status);
}
else
{
  console.log(error);
}
})
setTimeout(function(){
events.stopWatching();
}, 60000)
```

This is how the preceding code works:

1. At first, we get the event object by calling the method of the event namesake on a contract instance. This method takes two objects as arguments, which are used to filter events:
    - The first object is used to filter events by indexed return values: for example, `{'valueA': 1, 'valueB': [myFirstAddress, mySecondAddress]}`. By default, all filter values are set to `null`. This means that they will match any event of a given type sent from this contract.
    - The next object can contain three properties: `fromBlock` (the earliest block; by default, it is `"latest"`), `toBlock` (the latest block; by default, it is `"latest"`), and `address` (a list of addresses to only get logs from; by default, the contract address).

2. The `event` object exposes three methods: `get`, `watch`, and `stopWatching`. `get` is used to get all the events in the block range. `watch` is like `get` but it watches for changes after getting the events. And `stopWatching` can be used to stop watching for changes.
3. Then, we have the `allEvents` method of the contract instance. It is used to retrieve all the events of a contract.
4. Every event is represented by an object that contains the following properties:
    - `args`: An object with the arguments from the event
    - `event`: A string representing the event name
    - `logIndex`: An integer representing the log index position in the block
    - `transactionIndex`: An integer representing the transactions the index position log was created from
    - `transactionHash`: A string representing the hash of the transactions this log was created from
    - `address`: A string representing the address from which this log originated
    - `blockHash`: A string representing the hash of the block where this log was in; `null` when its pending
    - `blockNumber`: The block number where this log was in; `null` when its pending

web3.js provides a `web3.eth.filter` API to retrieve and watch for events. You can use this API, but the earlier method's way of handling events is much easier. You can learn more about it at `https://github.com/ethereum/wiki/wiki/JavaScript-API#web3ethfilter`.

# Building a client for an ownership contract

In the previous chapter, we wrote Solidity code for the ownership contract, and in both the previous chapter and this chapter, we learned web3.js and how to invoke the methods of the contract using web3.js. Now, it's time to build a client for our smart contract so that users can use it easily.

We will build a client where a user selects a file and enters owner details and then clicks on **Submit** to broadcast a transaction to invoke the contract's `set` method with the file hash and the owner's details. Once the transaction is successfully broadcasted, we will display the transaction hash. The user will also be able to select a file and get the owner's details from the smart contract. The client will also display the recent `set` transactions mined in real time.

We will use sha1.js to get the hash of the file on the frontend, jQuery for DOM manipulation, and Bootstrap 4 to create a responsive layout. We will use express.js and web3.js on the backend. We will use socket.io so that the backend pushes recently mined transactions to the frontend without the frontend requesting for data after every equal interval of time.

web3.js can be used in the frontend. But for this application, it will be a security risk; that is, we are using accounts stored in geth and exposing the geth node URL to the frontend, which will put the ether stored in those accounts at risk.

## The project structure

In the exercise files of this chapter, you will find two directories: `Final` and `Initial`. `Final` contains the final source code of the project, whereas `Initial` contains the empty source code files and libraries to get started with building the application quickly.

To test the `Final` directory, you will need to run `npm install` inside it and replace the hardcoded contract address in `app.js` with the contract address you got after deploying the contract. Then, run the app using the `node app.js` command inside the `Final` directory.

In the `Initial` directory, you will find a `public` directory and two files named `app.js` and `package.json`. `package.json` contains the backend dependencies of our app, and `app.js` is where you will place the backend source code.

The `public` directory contains files related to the frontend. Inside `public/css`, you will find `bootstrap.min.css`, which is the Bootstrap library; inside `public/html`, you will find `index.html`, where you will place the HTML code of our app; and in the `public/js` directory, you will find JS files for jQuery, sha1, and socket.io. Inside `public/js`, you will also find a `main.js` file, where you will place the frontend JS code of our app.

# Building the backend

Let's first build the backend of the app. First of all, run `npm install` inside the `Initial` directory to install the required dependencies for our backend. Before we get into coding the backend, make sure geth is running with rpc enabled. If you are running geth on a private network, then make sure mining is also enabled. Finally, make sure that account 0 exists and is unlocked. You can run geth on a private network with rpc and mining enabled and also unlocking account 0:

```
geth --dev --mine --rpc --unlock=0
```

One final thing you need to do before getting started with coding is deploy the ownership contract using the code we saw in the previous chapter and copy the contract address.

Now let's create a single server, which will serve the HTML to the browser and also accept `socket.io` connections:

```
var express = require("express");
var app = express();
var server = require("http").createServer(app);
var io = require("socket.io")(server);
server.listen(8080);
```

Here, we are integrating both the `express` and `socket.io` servers into one server running on port `8080`.

Now let's create the routes to serve the static files and also the home page of the app. Here is the code to do this:

```
app.use(express.static("public"));
app.get("/", function(req, res){
  res.sendFile(__dirname + "/public/html/index.html");
})
```

Here, we are using the `express.static` middleware to serve static files. We are asking it to find static files in the `public` directory.

Now let's connect to the `geth` node and also get a reference to the deployed contract so that we can send transactions and watch for events. Here is the code to do this:

```
var Web3 = require("web3");

web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));

var proofContract =
web3.eth.contract([{"constant":false,"inputs":[{"name":"fileHash","type":"s
tring"}],"name":"get","outputs":[{"name":"timestamp","type":"uint256"},{"na
me":"owner","type":"string"}],"payable":false,"type":"function"},{"constant
":false,"inputs":[{"name":"owner","type":"string"},{"name":"fileHash","type
":"string"}],"name":"set","outputs":[],"payable":false,"type":"function"},{
"anonymous":false,"inputs":[{"indexed":false,"name":"status","type":"bool"}
,{"indexed":false,"name":"timestamp","type":"uint256"},{"indexed":false,"na
me":"owner","type":"string"},{"indexed":false,"name":"fileHash","type":"str
ing"}],"name":"logFileAddedStatus","type":"event"}]);

var proof = proofContract.at("0xf7f02f65d5cd874d180c3575cb8813a9e7736066");
```

The code is self-explanatory. Just replace the contract address with the one you got.

Now let's create routes to broadcast transactions and get information about a file. Here is the code to do this:

```
app.get("/submit", function(req, res){
var fileHash = req.query.hash;
var owner = req.query.owner;
proof.set.sendTransaction(owner, fileHash, {
from: web3.eth.accounts[0],
}, function(error, transactionHash){
if (!error)
{
  res.send(transactionHash);
}
else
{
  res.send("Error");
}
})
})
app.get("/getInfo", function(req, res){
var fileHash = req.query.hash;
var details = proof.get.call(fileHash);
res.send(details);
})
```

Here, the /submit route is used to create and broadcast transactions. Once we get the transaction hash, we send it to the client. We are not doing anything to wait for the transaction to mine. The /getInfo route calls the get method of the contract on the node itself instead of creating a transaction. It simply sends back whatever response it got.

Now let's watch for the events from the contract and broadcast it to all the clients. Here is the code to do this:

```
proof.logFileAddedStatus().watch(function(error, result){
if(!error)
{
  if(result.args.status == true)
  {
    io.send(result);
  }
}
})
```

Here, we check whether the status is true, and if it's true, only then do we broadcast the event to all the connected socket.io clients.

# Building the frontend

Let's begin with the HTML of the app. Put this code in the index.html file:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
        <link rel="stylesheet" href="/css/bootstrap.min.css">
    </head>
    <body>
        <div class="container">
            <div class="row">
                <div class="col-md-6 offset-md-3 text-xs-center">
                    <br>
                    <h3>Upload any file</h3>
                    <br>
                    <div>
                        <div class="form-group">
                            <label class="custom-file text-xs-left">
                                <input type="file" id="file" class="custom-
file-input">
                                <span class="custom-file-control"></span>
                            </label>
                        </div>
                        <div class="form-group">
                            <label for="owner">Enter owner name</label>
                            <input type="text" class="form-control"
id="owner">
                        </div>
                        <button onclick="submit()" class="btn btn-
primary">Submit</button>
                        <button onclick="getInfo()" class="btn btn-
primary">Get Info</button>
                        <br><br>
                        <div class="alert alert-info" role="alert"
id="message">
                            You can either submit file's details or get
information about it.
                        </div>
                    </div>
                </div>
            </div>
            <div class="row">
                <div class="col-md-6 offset-md-3 text-xs-center">
                    <br>
                    <h3>Live Transactions Mined</h3>
                    <br>
                    <ol id="events_list">No Transaction Found</ol>
                </div>
            </div>
        </div>
        <script type="text/javascript" src="/js/sha1.min.js"></script>
        <script type="text/javascript" src="/js/jquery.min.js"></script>
        <script type="text/javascript" src="/js/socket.io.min.js"></script>
        <script type="text/javascript" src="/js/main.js"></script>
    </body>
</html>
```

Here is how the code works:

1. At first, we display Bootstrap's file input field so that the user can select a file.
2. Then, we display a text field where the user can enter the owner's details.
3. Then, we have two buttons. The first one is to store the file hash and the owner's details in the contract, and the second button is to get information on the file from the contract. Clicking on the **Submit** button triggers the `submit()` method, whereas clicking on the **Get Info** button triggers the `getInfo()` method.

4. Then, we have an alert box to display messages.
5. Finally, we display an ordered list to display the transactions of the contract that gets mined while the user is on the page.

Now let's write the implementation for the `getInfo()` and `submit()` methods, establish a `socket.io` connect with the server, and listen for `socket.io` messages from the server. Here is the code to this. Place this code in the `main.js` file:

```
    function submit()
    {
      var file = document.getElementById("file").files[0];
      if(file)
    {
     var owner = document.getElementById("owner").value;
     if(owner == "")
    {
     alert("Please enter owner name");
    }
   else
   {
    var reader = new FileReader();
    reader.onload = function (event) {
    var hash = sha1(event.target.result);
    $.get("/submit?hash=" + hash + "&owner=" + owner, function(data){
    if(data == "Error")
    {
      $("#message").text("An error occured.");
    }
    else
    {
      $("#message").html("Transaction hash: " + data);
    }
    });
    };
    reader.readAsArrayBuffer(file);
     }
  }
    else
    {
      alert("Please select a file");
    }
  }
  function getInfo()
  {
    var file = document.getElementById("file").files[0];
    if(file)
    {
```

```
    var reader = new FileReader();
    reader.onload = function (event) {
    var hash = sha1(event.target.result);
    $.get("/getInfo?hash=" + hash, function(data){
    if(data[0] == 0 && data[1] == "")
    {
      $("#message").html("File not found");
    }
    else
    {
      $("#message").html("Timestamp: " + data[0] + " Owner: " + data[1]);
    }
  });
};
reader.readAsArrayBuffer(file);
}
else
  {
    alert("Please select a file");
  }
}
var socket = io("http://localhost:8080");
socket.on("connect", function () {
socket.on("message", function (msg) {
if($("#events_list").text() == "No Transaction Found")
{
    $("#events_list").html("<li>Txn Hash: " + msg.transactionHash +
"nOwner: " + msg.args.owner + "nFile Hash: " + msg.args.fileHash +
"</li>");
}
else
{
  $("#events_list").prepend("<li>Txn Hash: " + msg.transactionHash +
"nOwner: " + msg.args.owner + "nFile Hash: " + msg.args.fileHash +
"</li>");
}
  });
});
```

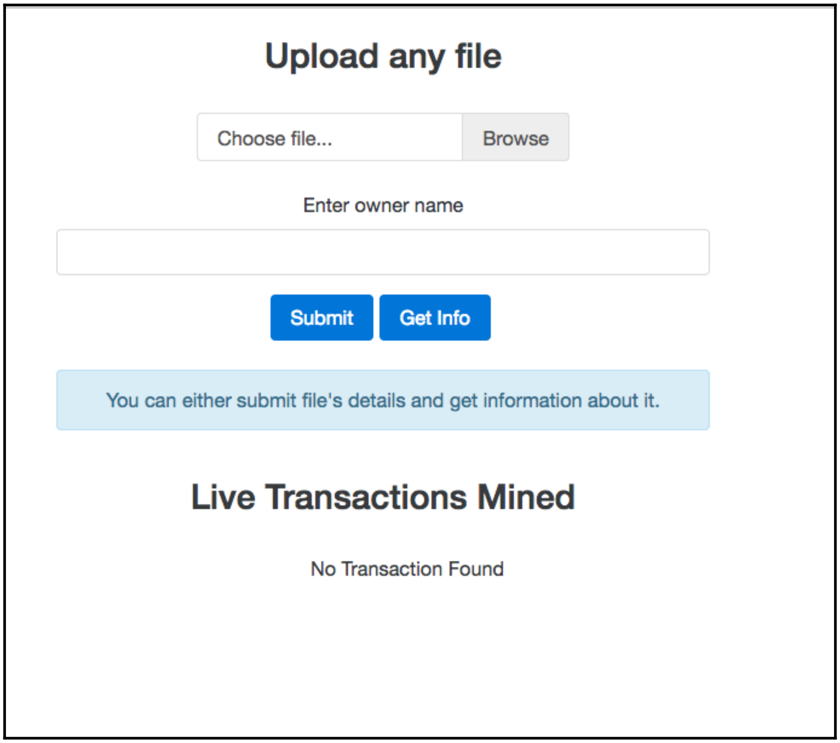
This is how the preceding code works:

1. At first, we defined the `submit()` method. In the `submit` method, we make sure that a file is selected and the text field is not empty. Then, we read the content of the file as an array buffer and pass the array buffer to the `sha1()` method exposed by sha1.js to get the hash of content inside the array buffer. Once we have the hash, we use jQuery to make an AJAX request to the `/submit` route and then we display the transaction hash in the alert box.
2. We define the `getInfo()` method next. It first makes sure that a file is selected. Then, it generates the hash like the one it generated earlier and makes a request to the `/getInfo` endpoint to get information about that file.
3. Finally, we establish a `socket.io` connection using the `io()` method exposed by the `socket.io` library. Then, we wait for the connect event to the trigger, which indicates that a connection has been established. After the connection is established, we listen for messages from the server and display the details about the transactions to the user.

We aren't storing the file in the Ethereum blockchain because storing files is very expensive as it requires a lot of gas. For our case, we actually don't need to store files because nodes in the network will be able to see the file; therefore, if the users want to keep the file content secret, then they won't be able to. Our application's purpose is just to prove ownership of a file, not to store and serve the file like a cloud service.

# Testing the client

Now run the `app.js` node to run the application server. Open you favorite browser and visit `http://localhost:8080/`. You will see this output in the browser:



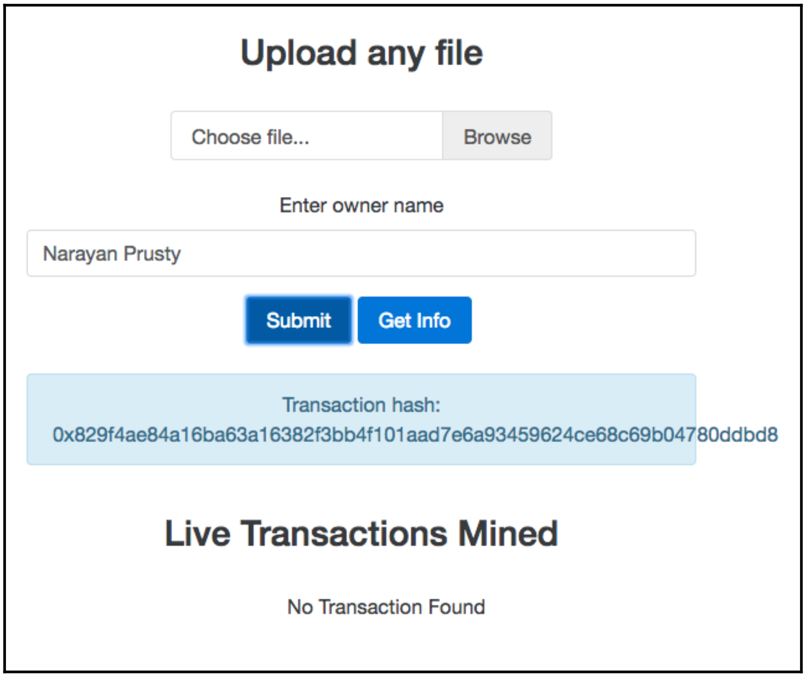
Upload any file

You can either submit file's details and get information about it.

Live Transactions Mined

No Transaction Found

Now select a file and enter the owner's name and click on **Submit**. The screen will change to this:



Upload any file

Enter owner name

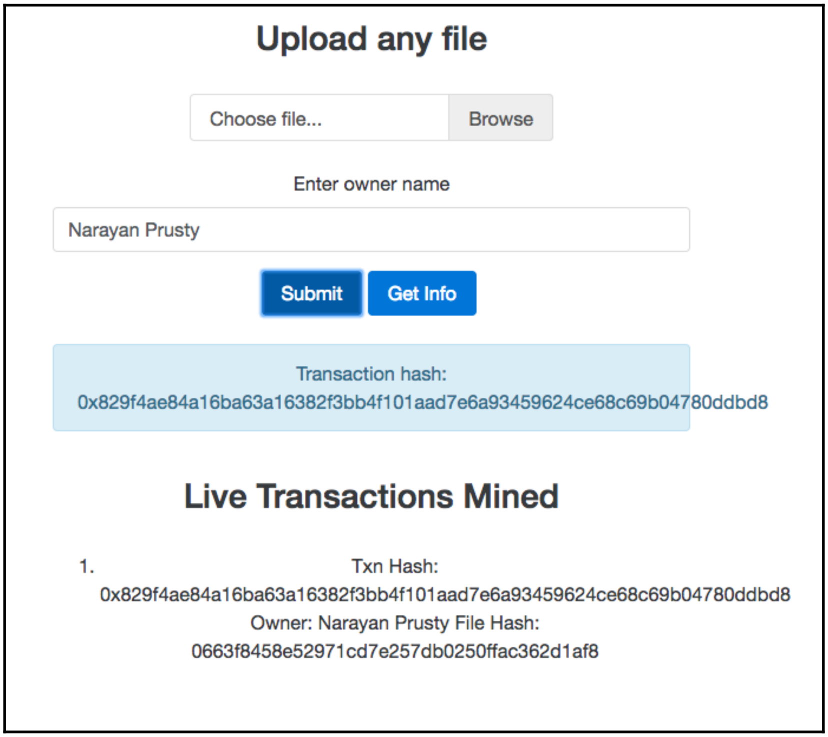
Narayan Prusty

Transaction hash:
0x829f4ae84a16ba63a16382f3bb4f101aad7e6a93459624ce68c69b04780ddbd8

Live Transactions Mined

No Transaction Found

Here, you can see that the transaction hash is displayed. Now wait until the transaction is mined. Once the transaction is mined, you will be able to see the transaction in the live transactions list. Here is how the screen would look:



Upload any file

Enter owner name

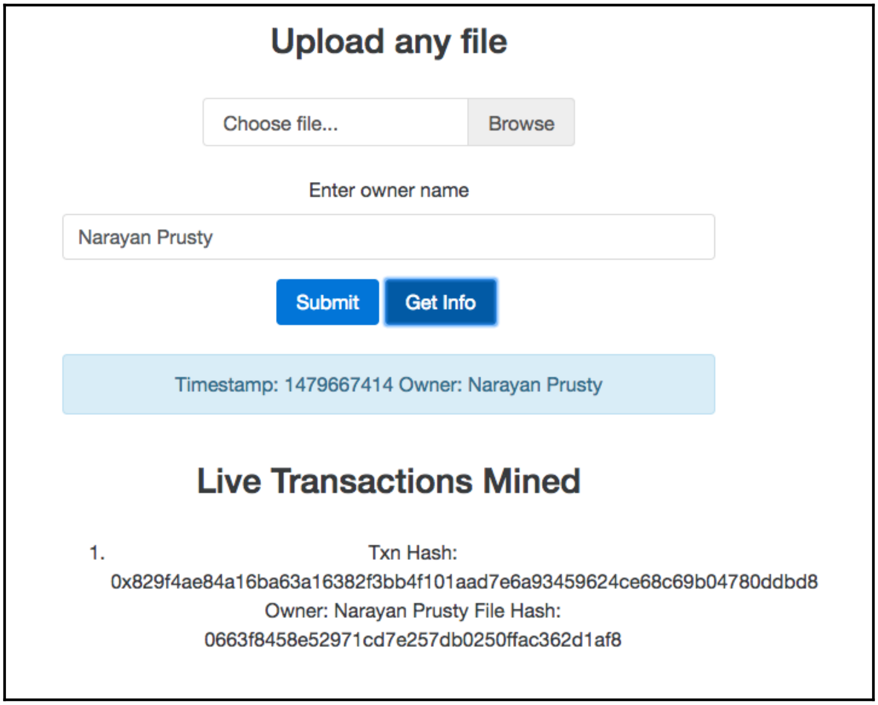
Narayan Prusty

Transaction hash:
0x829f4ae84a16ba63a16382f3bb4f101aad7e6a93459624ce68c69b04780ddbd8

Live Transactions Mined

1. Txn Hash:
0x829f4ae84a16ba63a16382f3bb4f101aad7e6a93459624ce68c69b04780ddbd8
Owner: Narayan Prusty File Hash:
0663f8458e52971cd7e257db0250ffac362d1af8

Now select the same file again and click on the **Get Info** button. You will see this output:



Upload any file

Enter owner name

Narayan Prusty

Timestamp: 1479667414 Owner: Narayan Prusty

Live Transactions Mined

1. Txn Hash: 0x829f4ae84a16ba63a16382f3bb4f101aad7e6a93459624ce68c69b04780ddbd8 Owner: Narayan Prusty File Hash: 0663f8458e52971cd7e257db0250ffac362d1af8

Here, you can see the timestamp and the owner's details. Now we have finished building the client for our first DApp.

# Summary

In this chapter, we first learned about the fundamentals of web3.js with examples. We learned how to connect to a node, basic APIs, sending various kinds of transactions, and watching for events. Finally, we built a proper production use client for our ownership contract. Now you will be comfortable with writing smart contracts and building UI clients for them in order to ease their use.

In the next chapter, we will build a wallet service, where users can create and manage Ethereum Wallets easily, and that too is offline. We will specifically use the LightWallet library to achieve this.

# 5
# Building a Wallet Service

A wallet service is used to send and receive funds. Major challenges for building a wallet service are security and trust. Users must feel that their funds are secure and the administrator of the wallet service doesn't steal their funds. The wallet service we will build in this chapter will tackle both these issues.

In this chapter, we'll cover the following topics:

- Difference between online and offline wallets
- Using hooked-web3-provider and ethereumjs-tx to make it easier to create and sign transactions using accounts not managed by an Ethereum node
- Understanding what a HD wallet is and also its uses
- Creating an HD wallet and a transaction signer using `lightwallet.js`
- Building a wallet service

## Difference between online and offline wallets

A wallet is a collection of accounts, and an account is a combination of an address and its associated private key.

A wallet is said to be an online wallet when it is connected to the Internet. For example, wallets stored in geth, any website/database, and so on are called online wallets. Online wallets are also called hot wallets, web wallets, hosted wallets, and so on. Online wallets are not recommended at least when storing large amounts of ether or storing ether for a long time because they are risky. Also, depending on where the wallet is stored, it may require trusting a third party.

For example, most of the popular wallet services store the private keys of the wallets with themselves and allow you to access the wallet via an e-mail and password, so basically, you don't have actual access to the wallet, and if they want, they can steal the funds in the wallets.

A wallet is said to be an offline wallet when it is not connected to the Internet. For example, wallets stored in pen drives, papers, text files, and so on. Offline wallets are also called cold wallets. Offline wallets are more secure than online wallets because to steal funds, someone will need physical access to the storage. The challenges with offline storage is that you need to find a location that you won't delete accidentally or forget, or nobody else can have access to. Many people store the wallet in paper and keep the paper in a safe locker if they want to hold some funds safely for a very long time. If you want to frequently send funds from your account, then you can store it in a password-protected pen drive and also in a safe locker. It is a little risker to store wallets in a digital device only because digital devices can corrupt anytime and you may lose access to your wallet; that's why along with storing in a pen drive, you should also put it in a safe locker. You can also find a better solution depending on your needs, but just make sure it's safe and that you accidentally don't lose access to it.

# hooked-web3-provider and ethereumjs-tx libraries

Until now, all the examples of Web3.js library's `sendTransaction()` method we saw were using the `from` address that's present in the Ethereum node; therefore, the Ethereum node was able to sign the transactions before broadcasting. But if you have the private key of a wallet stored somewhere else, then geth cannot find it. Therefore, in this case, you will need to use the `web3.eth.sendRawTransaction()` method to broadcast transactions.

`web3.eth.sendRawTransaction()` is used to broadcast raw transactions, that is, you will have to write code to create and sign raw transactions. The Ethereum node will directly broadcast it without doing anything else to the transaction. But writing code to broadcast transactions using `web3.eth.sendRawTransaction()` is difficult because it requires generating the data part, creating raw transactions, and also signing the transactions.

The Hooked-Web3-Provider library provides us with a custom provider, which communicates with geth using HTTP; but the uniqueness of this provider is that it lets us sign the `sendTransaction()` calls of contract instances using our keys. Therefore, we don't need to create data part of the transactions anymore. The custom provider actually overrides the implementation of the `web3.eth.sendTransaction()` method. So basically, it lets us sign both the `sendTransaction()` calls of contract instances and also the `web3.eth.sendTransaction()` calls. The `sendTransaction()` method of contract instances internally generate data of the transaction and calls `web3.eth.sendTransaction()` to broadcast the transaction.

EthereumJS is a collection of those libraries related to Ethereum. ethereumjs-tx is one of those that provide various APIs related to transactions. For example, it lets us create raw transactions, sign the raw transactions, check whether transactions are signed using proper keys or not, and so on.

Both of these libraries are available for Node.js and client-side JavaScript. Download the Hooked-Web3-Provider from `https://www.npmjs.com/package/hooked-web3-provider`, and download ethereumjs-tx from `https://www.npmjs.com/package/ethereumjs-tx`.

At the time of writing this book, the latest version of Hooked-Web3-Provider is 1.0.0 and the latest version of ethereumjs-tx is 1.1.4.

Let's see how to use these libraries together to send a transaction from an account that's not managed by geth.

```
var provider = new HookedWeb3Provider({
 host: "http://localhost:8545",
 transaction_signer: {
  hasAddress: function(address, callback){
   callback(null, true);
  },
  signTransaction: function(tx_params, callback){
   var rawTx = {
          gasPrice: web3.toHex(tx_params.gasPrice),
          gasLimit: web3.toHex(tx_params.gas),
          value: web3.toHex(tx_params.value)
          from: tx_params.from,
          to: tx_params.to,
          nonce: web3.toHex(tx_params.nonce)
      };

   var privateKey =
EthJS.Util.toBuffer('0x1a56e47492bf3df9c9563fa7f66e4e032c661de9d68c3f36f358
e6bc9a9f69f2', 'hex');
   var tx = new EthJS.Tx(rawTx);
```

```
      tx.sign(privateKey);

      callback(null, tx.serialize().toString('hex'));
     }
   }
  });

  var web3 = new Web3(provider);

  web3.eth.sendTransaction({
   from: "0xba6406ddf8817620393ab1310ab4d0c2deda714d",
   to: "0x2bdbec0ccd70307a00c66de02789e394c2c7d549",
   value: web3.toWei("0.1", "ether"),
   gasPrice: "20000000000",
   gas: "21000"
  }, function(error, result){
   console.log(error, result)
  })
```

Here is how the code works:

1. At first, we created a HookedWeb3Provider instance. This is provided by the Hooked-Web3-Provider library. This constructor takes an object that has two properties, which must be provided. host is the HTTP URL of the node and transaction_signer is an object that the custom provider communicates with to get the transaction signed.
2. The transaction_signer object has two properties: hasAddress and signTransaction. hasAddress is invoked to check whether the transaction can be signed, that is, to check whether the transaction signer has the private key of the from address account. This method receives the address and a callback. The callback should be called with the first argument as an error message and the second argument as false if the private key of the address is not found. And if the private key is found, the first argument should be null, and the second argument should be true.

3. If the private key for the address is found, then the custom provider invokes the `signTransaction` method to get the transaction signed. This method has two parameters, that is, the transactions parameters and a callback. Inside the method, at first, we convert the transaction parameters to raw transaction parameters, that is, the raw transaction parameters values are encoded as hexadecimal strings. Then we create a buffer to hold the private key. The buffer is created using the `EthJS.Util.toBuffer()` method, which is part of the `ethereumjs-util` library. The `ethereumjs-util` library is imported by the `ethereumjs-tx` library. We then create a raw transaction and sign it, after which we serialize and convert it to hexadecimal strings. Finally, we need to provide the hexadecimal string of the signed raw transaction to the custom provider using the callback. In case there is an error inside the method, then the first argument of the callback should be an error message.
4. Now the custom provider takes the raw transactions and broadcasts it using `web3.eth.sendRawTransaction().`
5. Finally, we call the `web3.eth.sendTransaction` function to send some ether to another account. Here, we need to provide all the transaction parameters except `nonce` because the custom provider can calculate nonce. Earlier, many of these were optional because we were leaving it to the Ethereum node to calculate them, but now as we are signing it ourselves, we need to provide all of them. The `gas` is always 21,000 when the transaction doesn't have any data associated with it.

TIP

**What about the public key?**
In the preceding code, nowhere did we mention anything about the public key of the signing address. You must be wondering how a miner will verify the authenticity of a transaction without the public key. Miners use a unique property of ECDSA, which allows you to calculate the public key from the message and signature. In a transaction, the message indicates the intention of the transaction, and the signature is used to find whether the message is signed using the correct private key. This is what makes ECDSA so special. ethereumjs-tx provides an API to verify transactions.

# What is a hierarchical deterministic wallet?

A hierarchical deterministic wallet is a system of deriving addresses and keys from a single starting point called a seed. Deterministic indicates that for the same seed, the same addresses and keys will be generated, and hierarchical indicates that the addresses and keys will be generated in the same order. This makes it easier to back up and store multiple accounts, as you just have to store the seed, not the individual keys and addresses.

**Why will users need multiple accounts?**
You must be wondering why users will need multiple accounts. The reason is to hide their wealth. The balance of accounts is available publicly in the blockchain. So, if user A shares an address with user B to receive some ether, then user B can check how much ether is present in that address. Therefore, users usually distribute their wealth across various accounts.

There are various types of HD wallets, which differ in terms of seed format and the algorithm to generate addresses and keys, for instance, BIP32, Armory, Coinkite, Coinb.in, and so on.

**What are BIP32, BIP44, and BIP39?**
A **Bitcoin Improvement Proposal** (**BIP**) is a design document providing information to the Bitcoin community, or describing a new feature for Bitcoin or its processes or environment. The BIP should provide a concise technical specification of the feature and a rationale for the feature. At the time of writing this book, there are 152 BIPS (Bitcoin Improvement Proposals). BIP32 and BIP39 provide information about an algorithm to implement an HD wallet and mnemonic seed specification respectively. You can learn more about these at `https://github.com/bitcoin/bips`.

# Introduction to key derivation functions

Asymmetric cryptography algorithms define the nature of their keys and how the keys should be generated because the keys need to be related. For example, the RSA key generation algorithm is deterministic.

Symmetric cryptography algorithms only define key sizes. It's up to us to generate the keys. There are various algorithms to generate these keys. One such algorithm is KDF.

A **key derivation function** (**KDF**) is a deterministic algorithm to derive a symmetric key from some secret value (such as master key, password, or passphrase). There are various types of KDFs, such as bcrypt, crypt, PBKDF2, scrypt, HKDF, and so on. You can learn more about KDFs at `https://en.wikipedia.org/wiki/Key_derivation_function`.

To generate multiple keys from a single secret value, you can concatenate a number and increment it.

A password-based key derivation function takes a password and generates a symmetric key. Due to the fact that users usually use weak passwords, password-based key derivation functions are designed to be slower and take a lot of memory to make it difficult to launch brute force attacks and other kinds of attacks. Password-based key derivation functions are used widely because it's difficult to remember secret keys, and storing them somewhere is risky as it can be stolen. PBKDF2 is an example of a password-based key derivation function.

A master key or passphrase is difficult to be cracked using a brute force attack; therefore, in case you want to generate a symmetric key from a master key or passphrase, you can use a non-password-based key derivation function, such as HKDF. HKDF is much faster compared to PBKDF2.

**Why not just use a hash function instead of KDFs?**
The output of hash functions can be used as symmetric keys. So you must be wondering what is the need for KDFs. Well, if you are using a master key, passphrase, or a strong password, you can simply use a hash function. For example, HKDF simply uses a hash function to generate the key. But if you cannot guarantee that users will use a strong password, it's better to use a password derived hash function.

# Introduction to LightWallet

LightWallet is an HD wallet that implements BIP32, BIP39, and BIP44. LightWallet provides APIs to create and sign transactions or encrypt and decrypt data using the addresses and keys generated using it.

LightWallet API is divided into four namespaces, that is, `keystore`, `signing`, `encryption`, and `txutils`. `signing`, `encrpytion`, and `txutils` provide APIs to sign transactions, asymmetric cryptography, and create transactions respectively, whereas a `keystore` namespace is used to create a `keystore`, generated seed, and so on. `keystore` is an object that holds the seed and keys encrypted. The `keystore` namespace implements transaction signer methods that requires signing the `we3.eth.sendTransaction()` calls if we are using Hooked-Web3-Provider. Therefore the `keystore` namespace can automatically create and sign transactions for the addresses that it can find in it. Actually, LightWallet is primarily intended to be a signing provider for the Hooked-Web3-Provider.

A `keystore` instance can be configured to either create and sign transactions or encrypt and decrypt data. For signing transactions, it uses the `secp256k1` parameter, and for encryption and decryption, it uses the `curve25519` parameter.

The seed of LightWallet is a 12-word mnemonic, which is easy to remember yet difficult to hack. It cannot be any 12 words; instead, it should be a seed generated by LightWallet. A seed generated by LightWallet has certain properties in terms of selection of words and other things.

# HD derivation path

The HD derivation path is a string that makes it easy to handle multiple crypto currencies (assuming they all use the same signature algorithms), multiple blockchains, multiple accounts, and so on.

HD derivation path can have as many parameters as needed, and using different values for the parameters, we can produce different group of addresses and their associated keys.

By default, LightWallet uses the `m/0'/0'/0'` derivation path. Here, `/n'` is a parameter, and `n` is the parameter value.

Every HD derivation path has a `curve`, and `purpose`. `purpose` can be either `sign` or `asymEncrypt`. `sign` indicates that the path is used for signing transactions, and `asymEncrypt` indicates that the path is used for encryption and decryption. `curve` indicates the parameters of ECC. For signing, the parameter must be `secp256k1`, and for asymmetric encryption, the curve must be `curve25591` because LightWallet forces us to use these paramters due to their benefits in those purposes.

# Building a wallet service

Now we have learned enough theory about LightWallet, it's time to build a wallet service using LightWallet and hooked-web3-provider. Our wallet service will let users generate a unique seed, display addresses and their associated balance, and finally, the service will let users send ether to other accounts. All the operations will be done on the client side so that users can trust us easily. Users will either have to remember the seed or store it somewhere.

## Prerequisites

Before you start building the wallet service, make sure that you are running the geth development instance, which is mining, has the HTTP-RPC server enabled, allows client-side requests from any domain, and finally has account 0 unlocked. You can do all these by running this:

```
    geth --dev --rpc --rpccorsdomain "*" --rpcaddr "0.0.0.0" --rpcport
"8545" --mine --unlock=0
```

Here, `--rpccorsdomain` is used to allow certain domains to communicate with geth. We need to provide a list of domains space separated, such as `"http://localhost:8080 https://mySite.com *"`. It also supports the `*` wildcard character. `--rpcaddr` indicates to which IP address the geth server is reachable. The default for this is `127.0.0.1`, so if it's a hosted server, you won't be able to reach it using the public IP address of the server. Therefore, we changed it's value to `0.0.0.0`, which indicates that the server can be reached using any IP address.

## Project structure

In the exercise files of this chapter, you will find two directories, that is, `Final` and `Initial`. `Final` contains the final source code of the project, whereas `Initial` contains the empty source code files and libraries to get started with building the application quickly.

To test the `Final` directory, you will need to run `npm install` inside it and then run the app using the `node app.js` command inside the `Final` directory.

In the Initial directory, you will find a public directory and two files named app.js and package.json. package.json contains the backend dependencies. Our app, app.js, is where you will place the backend source code.

The public directory contains files related to the frontend. Inside public/css, you will find bootstrap.min.css, which is the bootstrap library. Inside public/html, you will find index.html, where you will place the HTML code of our app, and finally, in the public/js directory, you will find .js files for Hooked-Web3-Provider, web3js, and LightWallet. Inside public/js, you will also find a main.js file where you will place the frontend JS code of our app.

# Building the backend

Let's first build the backend of the app. First of all, run npm install inside the initial directory to install the required dependencies for our backend.

Here is the complete backend code to run an express service and serve the index.html file and static files:

```
var express = require("express");
var app = express();

app.use(express.static("public"));

app.get("/", function(req, res){
 res.sendFile(__dirname + "/public/html/index.html");
})

app.listen(8080);
```

The preceding code is self-explanatory.

# Building the frontend

Now let's build the frontend of the app. The frontend will consists of the major functionalities, that is, generating seed, displaying addresses of a seed, and sending ether.

Now let's write the HTML code of the app. Place this code in the index.html file:

```
<!DOCTYPE html>
 <html lang="en">
     <head>
```

```
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1, shrink-to-fit=no">
        <meta http-equiv="x-ua-compatible" content="ie=edge">
        <link rel="stylesheet" href="/css/bootstrap.min.css">
    </head>
    <body>
        <div class="container">
            <div class="row">
                <div class="col-md-6 offset-md-3">
                    <br>
                    <div class="alert alert-info" id="info" role="alert">
                          Create or use your existing wallet.
                    </div>
                    <form>
                        <div class="form-group">
                            <label for="seed">Enter 12-word seed</label>
                            <input type="text" class="form-control"
id="seed">
                        </div>
                        <button type="button" class="btn btn-primary"
onclick="generate_addresses()">Generate Details</button>
                        <button type="button" class="btn btn-primary"
onclick="generate_seed()">Generate New Seed</button>
                    </form>
                    <hr>
                    <h2 class="text-xs-center">Address, Keys and Balances
of the seed</h2>
                    <ol id="list">
                    </ol>
                    <hr>
                    <h2 class="text-xs-center">Send ether</h2>
                    <form>
                        <div class="form-group">
                            <label for="address1">From address</label>
                            <input type="text" class="form-control"
id="address1">
                        </div>
                        <div class="form-group">
                            <label for="address2">To address</label>
                            <input type="text" class="form-control"
id="address2">
                        </div>
                        <div class="form-group">
                            <label for="ether">Ether</label>
                            <input type="text" class="form-control"
id="ether">
                        </div>
                        <button type="button" class="btn btn-primary"
onclick="send_ether()">Send Ether</button>
                    </form>
                </div>
            </div>
        </div>

           <script src="/js/web3.min.js"></script>
           <script src="/js/hooked-web3-provider.min.js"></script>
        <script src="/js/lightwallet.min.js"></script>
        <script src="/js/main.js"></script>
    </body>
 </html>
```

Here is how the code works:

1. At first, we enqueue a Bootstrap 4 stylesheet.
2. Then we display an information box, where we will display various messages to the user.
3. And then we have a form with an input box and two buttons. The input box is used to enter the seed, or while generating new seed, the seed is displayed there.
4. The **Generate Details** button is used to display addresses and **Generate New Seed** is used to generate a new unique seed. When **Generate Details** is clicked, we call the `generate_addresses()` method, and when the **Generate New Seed** button is clicked, we call the `generate_seed()` method.
5. Later, we have an empty ordered list. Here, we will dynamically display the addresses, their balances, and associated private keys of a seed when a user clicks on the **Generate Details** button.
6. Finally, we have another form that takes a from address and a to address and the amount of ether to transfer. The from address must be one of the addresses that's currently displayed in the unordered list.

Now let's write the implementation of each of the functions that the HTML code calls. At first, let's write the code to generate a new seed. Place this code in the `main.js` file:

```
function generate_seed()
{
 var new_seed = lightwallet.keystore.generateRandomSeed();

 document.getElementById("seed").value = new_seed;

 generate_addresses(new_seed);
}
```

The `generateRandomSeed()` method of the `keystore` namespace is used to generate a random seed. It takes an optional parameter, which is a string that indicates the extra entropy.

Entropy is the randomness collected by an application for use in some algorithm or somewhere else that requires random data. Usually, entropy is collected from hardware sources, either pre-existing ones such as mouse movements or specially provided randomness generators.

To produce a unique seed, we need really high entropy. LightWallet is already built with methods to produce unique seeds. The algorithm LightWallet uses to produce entropy depends on the environment. But if you feel you can generate better entropy, you can pass the generated entropy to the `generateRandomSeed()` method, and it will get concatenated with the entropy generated by `generateRandomSeed()` internally.

After generating a random seed, we call the `generate_addresses` method. This method takes a seed and displays addresses in it. Before generating addresses, it prompts the user to ask how many addresses they want.

Here is the implementation of the `generate_addresses()` method. Place this code in the `main.js` file:

```
var totalAddresses = 0;

function generate_addresses(seed)
{
 if(seed == undefined)
 {
  seed = document.getElementById("seed").value;
 }

 if(!lightwallet.keystore.isSeedValid(seed))
 {
  document.getElementById("info").innerHTML = "Please enter a valid seed";
  return;
 }

 totalAddresses = prompt("How many addresses do you want to generate");

 if(!Number.isInteger(parseInt(totalAddresses)))
 {
  document.getElementById("info").innerHTML = "Please enter valid number of
addresses";
  return;
 }
```

```
    var password = Math.random().toString();

    lightwallet.keystore.createVault({
     password: password,
       seedPhrase: seed
    }, function (err, ks) {
       ks.keyFromPassword(password, function (err, pwDerivedKey) {
         if(err)
         {
          document.getElementById("info").innerHTML = err;
         }
         else
         {
          ks.generateNewAddress(pwDerivedKey, totalAddresses);
          var addresses = ks.getAddresses();

          var web3 = new Web3(new
   Web3.providers.HttpProvider("http://localhost:8545"));

          var html = "";

          for(var count = 0; count < addresses.length; count++)
          {
        var address = addresses[count];
        var private_key = ks.exportPrivateKey(address, pwDerivedKey);
        var balance = web3.eth.getBalance("0x" + address);

        html = html + "<li>";
        html = html + "<p><b>Address: </b>0x" + address + "</p>";
        html = html + "<p><b>Private Key: </b>0x" + private_key + "</p>";
        html = html + "<p><b>Balance: </b>" + web3.fromWei(balance, "ether") +
   " ether</p>";
           html = html + "</li>";
          }

          document.getElementById("list").innerHTML = html;
         }
       });
    });
   }
```

Here is how the code works:

1. At first, we have a variable named `totalAddresses`, which holds a number indicating the total number of addresses the user wants to generate.

2. Then we check whether the `seed` parameter is defined or not. If it's undefined, we fetch the seed from the input field. We are doing this so that the `generate_addressess()` method can be used to display the information seed while generating a new seed and also if the user clicks on the **Generate Details** button.

3. Then we validate the seed using the `isSeedValid()` method of the `keystore` namespace.
4. We then ask for the user's input regarding how many addresses they want to generate and display. And then we validate the input.
5. The private keys in the `keystore` namespace are always stored encrypted. While generating keys, we need to encrypt them, and while signing transactions, we need to decrypt the keys. The password for deriving a symmetric encryption key can be taken as input from the user or by supplying a random string as a password. For better user experience, we generate a random string and use it as the password. The symmetric key is not stored inside the `keystore` namespace; therefore, we need to generate the key from the password whenever we do operations related to the private key, such as generating keys, accessing keys, and so on.
6. Then we use the `createVault` method to create a `keystore` instance. `createVault` takes an object and a callback. The object can have four properties: `password`, `seedPharse`, `salt`, and `hdPathString`. `password` is compulsory, and everything else is optional. If we don't provide a `seedPharse`, it will generate and use a random seed. `salt` is concatenated to the password to increase the security of the symmetric key as the attacker has to also find the salt along with the password. If the salt is not provided, it's randomly generated. The `keystore` namespace holds the salt unencrypted. `hdPathString` is used to provide the default derivation path for the `keystore` namespace, that is, while generating addresses, signing transactions, and so on. If we don't provide a derivation path, then this derivation path is used. If we don't provide `hdPathString`, then the default value is `m/0'/0'/0'`. The default purpose of this derivation path is `sign`. You can create new derivation paths or overwrite the purpose of derivation paths present using the `addHdDerivationPath()` method of a `keystore` instance. You can also change the default derivation path using the `setDefaultHdDerivationPath()` method of a `keystore` instance. Finally, once the `keystore` namespace is created, the instance is returned via the callback. So here, we created a `keystore` using a password and seed only.

7. Now we need to generate the number of addresses and their associated keys the user needs. As we can generate millions of addresses from a seed, `keystore` doesn't generate any address until we want it to because it doesn't know how many addresses we want to generate. After creating the `keystore`, we generate the symmetric key from the password using the `keyFromPassword` method. And then we call the `generateNewAddress()` method to generate addresses and their associated keys.

8. `generateNewAddress()` takes three arguments: password derived key, number of addresses to generate, and derivation path. As we haven't provided a derivation path, it uses the default derivation path of the keystore. If you call `generateNewAddress()` multiple times, it resumes from the address it created in the last call. For example, if you call this method twice, each time generating two addresses, you will have the first four addresses.
9. Then we use `getAddresses()` to get all the addresses stored in the `keystore`.
10. We decrypt and retrieve private keys of the addresses using the `exportPrivateKey` method.
11. We use `web3.eth.getBalance()` to get balances of the address.
12. And finally, we display all the information inside the unordered list.

Now we know how to generate the address and their private keys from a seed. Now let's write the implementation of the `send_ether()` method, which is used to send ether from one of the addresses generated from the seed.

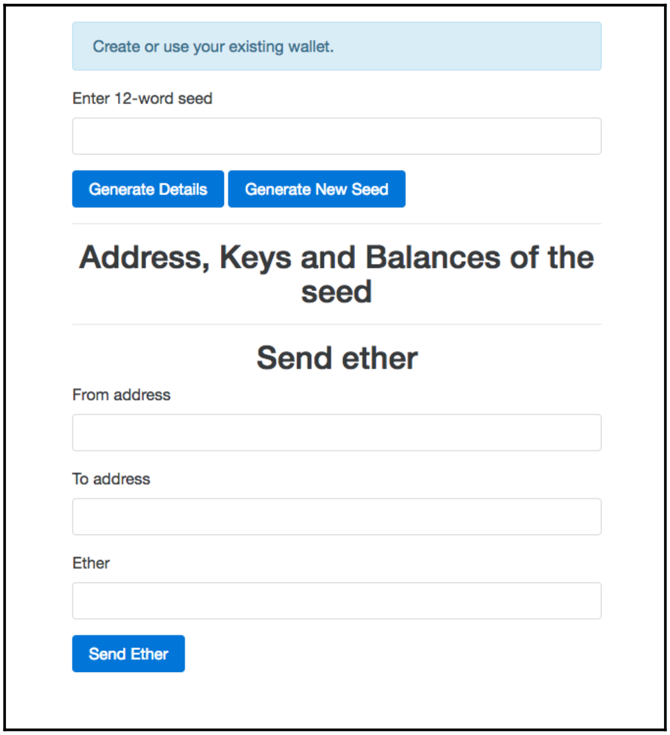
Here is the code for this. Place this code in the `main.js` file:

```
function send_ether()
{
 var seed = document.getElementById("seed").value;

 if(!lightwallet.keystore.isSeedValid(seed))
 {
  document.getElementById("info").innerHTML = "Please enter a valid seed";
  return;
 }

 var password = Math.random().toString();

 lightwallet.keystore.createVault({
  password: password,
    seedPhrase: seed
 }, function (err, ks) {
    ks.keyFromPassword(password, function (err, pwDerivedKey) {
      if(err)
      {
       document.getElementById("info").innerHTML = err;
      }
      else
      {
       ks.generateNewAddress(pwDerivedKey, totalAddresses);

       ks.passwordProvider = function (callback) {
          callback(null, password);
       };

       var provider = new HookedWeb3Provider({
       host: "http://localhost:8545",
       transaction_signer: ks
    });

       var web3 = new Web3(provider);

       var from = document.getElementById("address1").value;
    var to = document.getElementById("address2").value;
       var value = web3.toWei(document.getElementById("ether").value,
"ether");

       web3.eth.sendTransaction({
        from: from,
        to: to,
        value: value,
        gas: 21000
       }, function(error, result){
        if(error)
        {
         document.getElementById("info").innerHTML = error;
        }
        else
        {
         document.getElementById("info").innerHTML = "Txn hash: " + result;
        }
       })
      }
    });
 });
}
```

Here, the code up and until generating addresses from the seed is self explanatory. After that, we assign a callback to the `passwordProvider` property of `ks`. This callback is invoked during transaction signing to get the password to decrypt the private key. If we don't provide this, LightWallet prompts the user to enter the password. And then, we create a `HookedWeb3Provider` instance by passing the `keystore` as the transaction signer. Now when the custom provider wants a transaction to be signed, it calls the `hasAddress` and `signTransactions` methods of `ks`. If the address to be signed is not among the generated addresses, `ks` will give an error to the custom provider. And finally, we send some ether using the `web3.eth.sendTransaction` method.

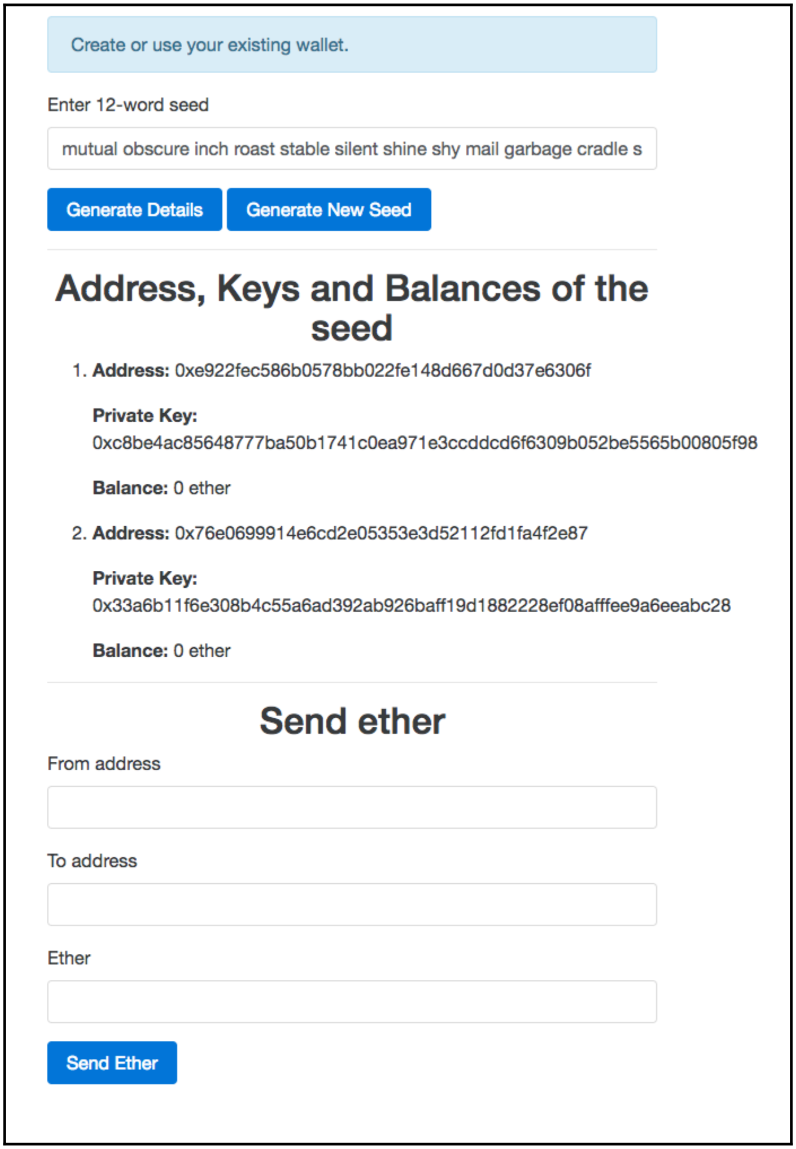
# Testing

Now that we have finished building our wallet service, let's test it to make sure it works as expected. First, run `node app.js` inside the initial directory, and then visit `http://localhost:8080` in your favorite browser. You will see this screen:



Create or use your existing wallet.

Now click on the **Generate New Seed** button to generate a new seed. You will be prompted to enter a number indicating the number of addresses to generate. You can provide any number, but for testing purposes, provide a number greater than 1. Now the screen will look something like this:



Create or use your existing wallet.

Enter 12-word seed

mutual obscure inch roast stable silent shine shy mail garbage cradle s

Address, Keys and Balances of the seed

1. **Address:** 0xe922fec586b0578bb022fe148d667d0d37e6306f

   **Private Key:**
   0xc8be4ac85648777ba50b1741c0ea971e3ccddcd6f6309b052be5565b00805f98

   **Balance:** 0 ether

2. **Address:** 0x76e0699914e6cd2e05353e3d52112fd1fa4f2e87

   **Private Key:**
   0x33a6b11f6e308b4c55a6ad392ab926baff19d1882228ef08afffee9a6eeabc28

   **Balance:** 0 ether

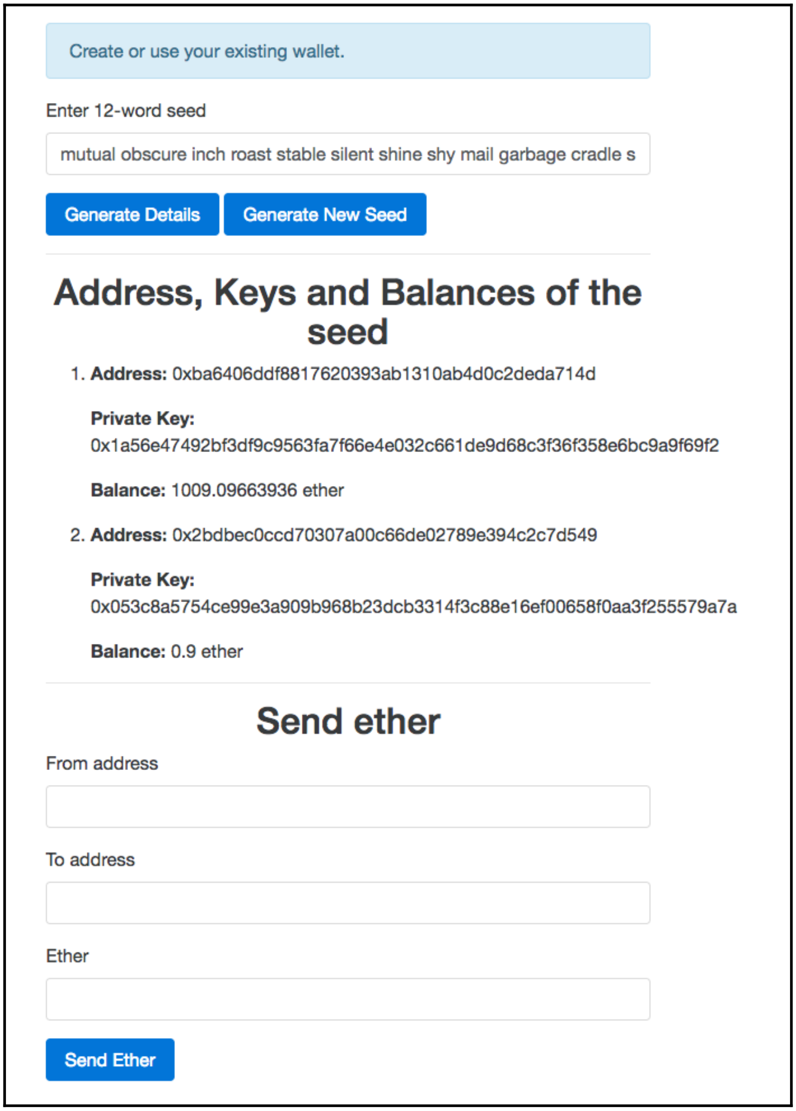
Send ether

From address

To address

Ether

Now to test sending ether, you need to send some ether to one of the generated addresses from the coinbase account. Once you have sent some ether to one of the generated addresses, click on the **Generate Details** button to refresh the UI, although it's not necessary to test sending ether using the wallet service. Make sure the same address is generated again. Now the screen will look something like this:



Create or use your existing wallet.

Enter 12-word seed

mutual obscure inch roast stable silent shine shy mail garbage cradle s

Address, Keys and Balances of the seed

1. **Address:** 0xba6406ddf8817620393ab1310ab4d0c2deda714d

   **Private Key:** 0x1a56e47492bf3df9c9563fa7f66e4e032c661de9d68c3f36f358e6bc9a9f69f2

   **Balance:** 1009.09663936 ether

2. **Address:** 0x2bdbec0ccd70307a00c66de02789e394c2c7d549

   **Private Key:** 0x053c8a5754ce99e3a909b968b23dcb3314f3c88e16ef00658f0aa3f255579a7a

   **Balance:** 0.9 ether

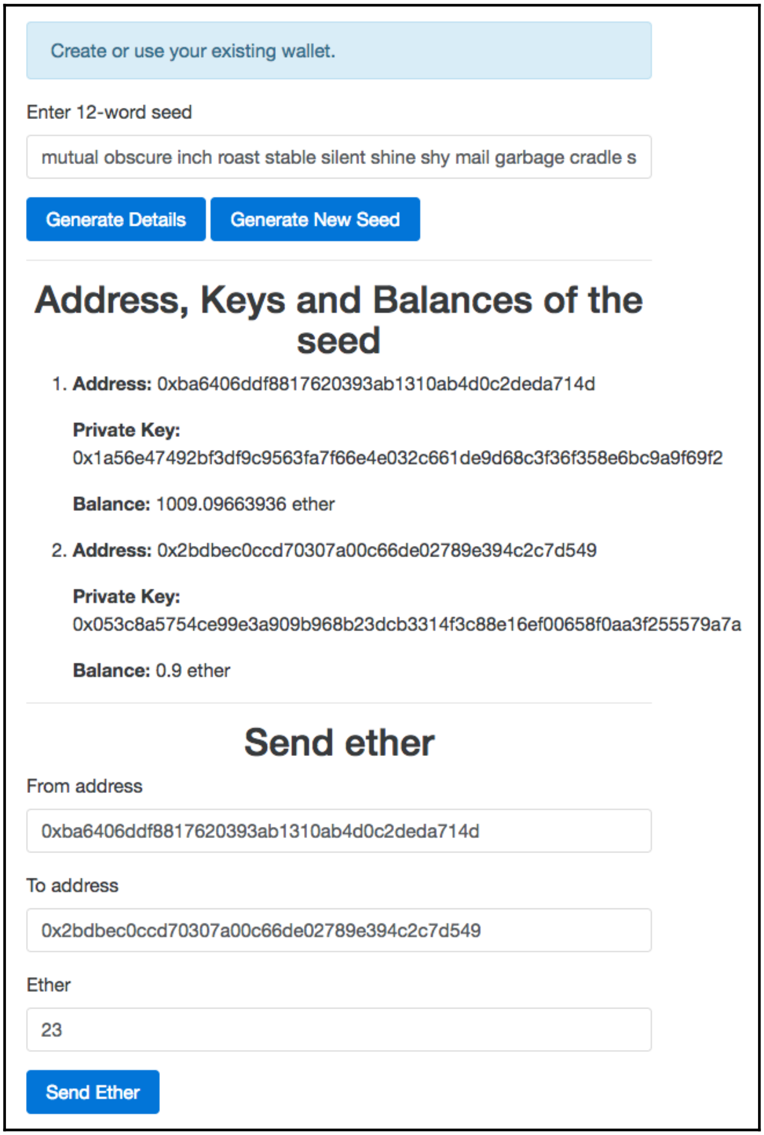
Send ether

From address

To address

Ether

Now in the from address field, enter the account address from the list that has the balance in the `from` address field. Then enter another address in the to address field. For testing purposes, you can enter any of the other addresses displayed. Then enter some ether amount that is less than or equal to the ether balance of the from address account. Now your screen will look something like this:



Create or use your existing wallet.

Enter 12-word seed

mutual obscure inch roast stable silent shine shy mail garbage cradle s

Address, Keys and Balances of the seed

1. **Address:** 0xba6406ddf8817620393ab1310ab4d0c2deda714d

   **Private Key:**
   0x1a56e47492bf3df9c9563fa7f66e4e032c661de9d68c3f36f358e6bc9a9f69f2

   **Balance:** 1009.09663936 ether

2. **Address:** 0x2bdbec0ccd70307a00c66de02789e394c2c7d549

   **Private Key:**
   0x053c8a5754ce99e3a909b968b23dcb3314f3c88e16ef00658f0aa3f255579a7a

   **Balance:** 0.9 ether

Send ether

From address

0xba6406ddf8817620393ab1310ab4d0c2deda714d

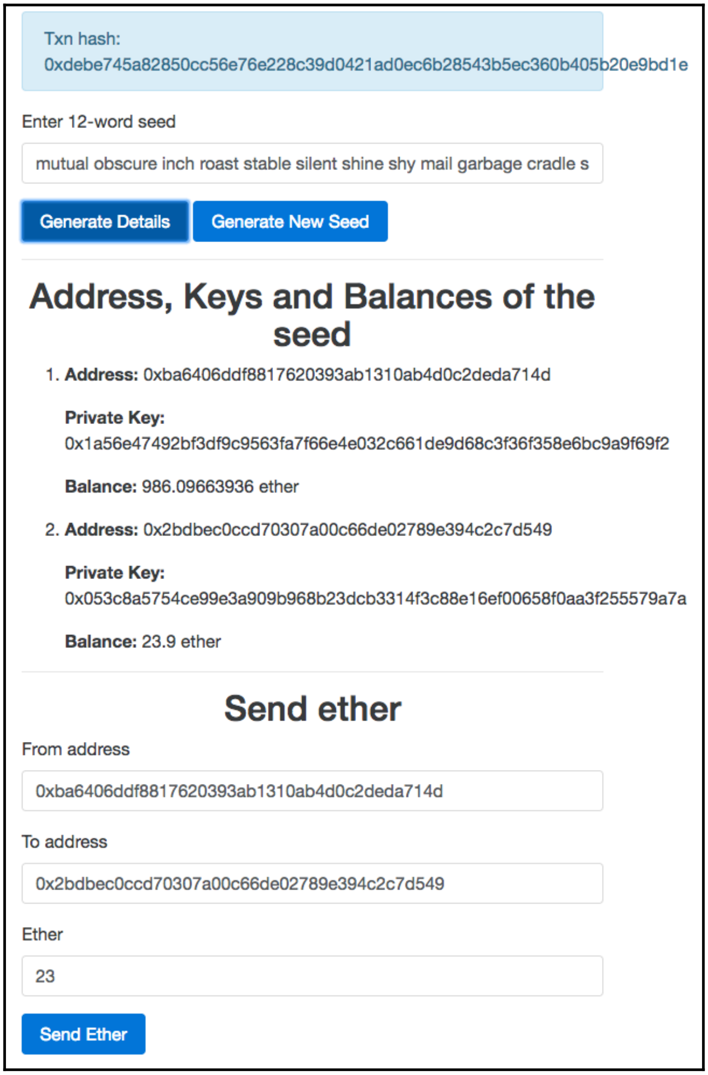
To address

0x2bdbec0ccd70307a00c66de02789e394c2c7d549

Ether

23

Now click on the **Send Ether** button, and you will see the transaction hash in the information box. Wait for sometime for it to get mined. Meanwhile, you can check whether the transactions got mined or not by clicking on the **Generate Details** button in a very short span of time. Once the transaction is mined, your screen will look something like this:



Txn hash:
0xdebe745a82850cc56e76e228c39d0421ad0ec6b28543b5ec360b405b20e9bd1e

Enter 12-word seed

mutual obscure inch roast stable silent shine shy mail garbage cradle s

Address, Keys and Balances of the seed

1. **Address:** 0xba6406ddf8817620393ab1310ab4d0c2deda714d

   **Private Key:**
   0x1a56e47492bf3df9c9563fa7f66e4e032c661de9d68c3f36f358e6bc9a9f69f2

   **Balance:** 986.09663936 ether

2. **Address:** 0x2bdbec0ccd70307a00c66de02789e394c2c7d549

   **Private Key:**
   0x053c8a5754ce99e3a909b968b23dcb3314f3c88e16ef00658f0aa3f255579a7a

   **Balance:** 23.9 ether

Send ether

From address

0xba6406ddf8817620393ab1310ab4d0c2deda714d

To address

0x2bdbec0ccd70307a00c66de02789e394c2c7d549

Ether

23

If everything goes the same way as explained, your wallet service is ready. You can actually deploy this service to a custom domain and make it available for use publicly. It's completely secure, and users will trust it.

# Summary

In this chapter, you learned about three important Ethereum libraries: Hooked-Web3-Provider, ethereumjs-tx, and LightWallet. These libraries can be used to manage accounts and sign transactions outside of the Ethereum node. While developing clients for most kinds of DApps, you will find these libraries useful.

And finally, we created a wallet service that lets users manage their accounts that share private keys or any other information related to their wallet with the backend of the service.

In the next chapter, we will build a platform to build and deploy smart contracts.

# 6
# Building a Smart Contract Deployment Platform

Some clients may need to compile and deploy contracts at runtime. In our proof-of-ownership DApp, we deployed the smart contract manually and hardcoded the contract address in the client-side code. But some clients may need to deploy smart contracts at runtime. For example, if a client lets schools record students' attendance in the blockchain, then it will need to deploy a smart contract every time a new school is registered so that each school has complete control over their smart contract. In this chapter, we will learn how to compile smart contracts using web3.js and deploy it using web3.js and EthereumJS.

In this chapter, we'll cover the following topics:

- Calculating the nonce of a transaction
- Using the transaction pool JSON-RPC API
- Generating data of a transaction for contract creation and method invocation
- Estimating the gas required by a transaction
- Finding the current spendable balance of an account
- Compiling smart contracts using solcjs
- Developing a platform to write, compile, and deploy smart contracts

## Calculating a transaction's nonce

For the accounts maintained by geth, we don't need to worry about the transaction nonce because geth can add the correct nonce to the transactions and sign them. While using accounts that aren't managed by geth, we need to calculate the nonce ourselves.

To calculate the nonce ourselves, we can use the `getTransactionCount` method provided by geth. The first argument should be the address whose transaction count we need and the second argument is the block until we need the transaction count. We can provide the `"pending"` string as the block to include transactions from the block that's currently being mined. As we discussed in an earlier chapter, geth maintains a transaction pool in which it keeps pending and queued transactions. To mine a block, geth takes the pending transactions from the transaction pool and starts mining the new block. Until the block is not mined, the pending transactions remain in the transaction pool and once mined, the mined transactions are removed from the transaction pool. The new incoming transactions received while a block is being mined are put in the transaction pool and are mined in the next block. So when we provide `"pending"` as the second argument while calling `getTransactionCount`, it doesn't look inside the transaction pool; instead, it just considers the transactions in the pending block.

So if you are trying to send transactions from accounts not managed by geth, then count the total number of transactions of the account in the blockchain and add it with the transactions pending in the transaction pool. If you try to use pending transactions from the pending block, then you will fail to get the correct nonce if transactions are sent to geth within a few seconds of the interval because it takes 12 seconds on average to include a transaction in the blockchain.

In the previous chapter, we relied on the hooked-web3-provider to add nonce to the transaction. Unfortunately, the hooked-web3-provider doesn't try to get the nonce the correct way. It maintains a counter for every account and increments it every time you send a transaction from that account. And if the transaction is invalid (for example, if the transaction is trying to send more ether than it has), then it doesn't decrement the counter. Therefore, the rest of the transactions from that account will be queued and never be mined until the hooked-web3-provider is reset, that is, the client is restarted. And if you create multiple instances of the hooked-web3-provider, then these instances cannot sync the nonce of an account with each other, so you may end up with the incorrect nonce. But before you add the nonce to the transaction, the hooked-web3-provider always gets the transaction count until the pending block and compares it with its counter and uses whichever is greater. So if the transaction from an account managed by the hooked-web3-provider is sent from another node in the network and is included in the pending block, then the hooked-web3-provider can see it. But the overall hooked-web3-provider cannot be relied on to calculate the nonce. It's great for quick prototyping of client-side apps and is fit to use in apps where the user can see and resend transactions if they aren't broadcasted to the network and the hooked-web3-provider is reset frequently. For example, in our wallet service, the user will frequently load the page, so a new hooked-web3-provider instance is created frequently. And if the transaction is not broadcasted, not valid, or not mined, then the user can refresh the page and resend transactions.

# Introducing solcjs

solcjs is a Node.js library and command-line tool that is used to compile solidity files. It doesn't use the solc command-line compiler; instead, it compiles purely using JavaScript, so it's much easier to install than solc.

Solc is the actual Solidity compiler. Solc is written in C++. The C++ code is compiled to JavaScript using emscripten. Every version of solc is compiled to JavaScript. At `https://github.com/ethereum/solc-bin/tree/gh-pages/bin`, you can find the JavaScript-based compilers of each solidity version. solcjs just uses one of these JavaScript-based compilers to compile the solidity source code. These JavaScript-based compilers can run in both browser and Node.js environments.

The browser Solidity uses these JavaScript-based compilers to compile the Solidity source code.

# Installing solcjs

solcjs is available as an npm package with the name `solc`. You can install the `solcjs` npm package locally or globally just like any other npm package. If this package is installed globally, then `solcjs`, a command-line tool, will be available. So, in order to install the command-line tool, run this command:

```
npm install -g solc
```

Now go ahead and run this command to see how to compile solidity files using the command-line compiler:

```
solcjs -help
```

We won't be exploring the solcjs command-line tool; instead, we will learn about the solcjs APIs to compile solidity files.

By default, solcjs uses compiler version matching as its version. For example, if you install solcjs version 0.4.8, then it will use the 0.4.8 compiler version to compile by default. solcjs can be configured to use some other compiler versions too. At the time of writing this, the latest version of solcjs is 0.4.8.

## solcjs APIs

solcjs provides a `compiler` method, which is used to compile solidity code. This method can be used in two different ways depending on whether the source code has any imports or not. If the source code doesn't have any imports, then it takes two arguments; that is, the first argument is solidity source code as a string and a Boolean indicating whether to optimize the byte code or not. If the source string contains multiple contracts, then it will compile all of them.

Here is an example to demonstrate this:

```
var solc = require("solc");
var input = "contract x { function g() {} }";
var output = solc.compile(input, 1); // 1 activates the optimiser
for (var contractName in output.contracts) {
    // logging code and ABI
    console.log(contractName + ": " +
output.contracts[contractName].bytecode);
    console.log(contractName + "; " +
JSON.parse(output.contracts[contractName].interface));
}
```

If your source code contains imports, then the first argument will be an object whose keys are filenames and values are the contents of the files. So whenever the compiler sees an import statement, it doesn't look for the file in the filesystem; instead, it looks for the file contents in the object by matching the filename with the keys. Here is an example to demonstrate this:

```
var solc = require("solc");
var input = {
    "lib.sol": "library L { function f() returns (uint) { return 7; } }",
    "cont.sol": "import 'lib.sol'; contract x { function g() { L.f(); } }"
};
var output = solc.compile({sources: input}, 1);
for (var contractName in output.contracts)
    console.log(contractName + ": " +
output.contracts[contractName].bytecode);
```

If you want to read the imported file contents from the filesystem during compilation or resolve the file contents during compilation, then the compiler method supports a third argument, which is a method that takes the filename and should return the file content. Here is an example to demonstrate this:

```
var solc = require("solc");
var input = {
    "cont.sol": "import 'lib.sol'; contract x { function g() { L.f(); } }"
};
function findImports(path) {
    if (path === "lib.sol")
        return { contents: "library L { function f() returns (uint) {
return 7; } }" }
    else
        return { error: "File not found" }
}
var output = solc.compile({sources: input}, 1, findImports);
for (var contractName in output.contracts)
    console.log(contractName + ": " +
output.contracts[contractName].bytecode);
```

## Using a different compiler version

In order to compile contracts using a different version of solidity, you need to use the `useVersion` method to get a reference of a different compiler. `useVersion` takes a string that indicates the JavaScript filename that holds the compiler, and it looks for the file in the `/node_modules/solc/bin` directory.

solcjs also provides another method called `loadRemoteVersion`, which takes the compiler filename that matches the filename in the `solc-bin/bin` directory of the `solc-bin` repository (`https://github.com/ethereum/solc-bin`) and downloads and uses it.

Finally, solcjs also provides another method called `setupMethods`, which is similar to `useVersion` but can load the compiler from any directory.

Here is an example to demonstrate all three methods:

```
var solc = require("solc");

var solcV047 = solc.useVersion("v0.4.7.commit.822622cf");
var output = solcV011.compile("contract t { function g() {} }", 1);

solc.loadRemoteVersion('soljson-v0.4.5.commit.b318366e', function(err,
solcV045) {
    if (err) {
        // An error was encountered, display and quit
    }

    var output = solcV045.compile("contract t { function g() {} }", 1);
});

var solcV048 = solc.setupMethods(require("/my/local/0.4.8.js"));
var output = solcV048.compile("contract t { function g() {} }", 1);

solc.loadRemoteVersion('latest', function(err, latestVersion) {
    if (err) {
        // An error was encountered, display and quit
    }
    var output = latestVersion.compile("contract t { function g() {} }",
1);
});
```

To run the preceding code, you need to first download the `v0.4.7.commit.822622cf.js` file from the `solc-bin` repository and place it in the `node_modules/solc/bin` directory. And then you need to download the compiler file of solidity version 0.4.8 and place it somewhere in the filesystem and point the path in the `setupMethods` call to that directory.

# Linking libraries

If your solidity source code references libraries, then the generated byte code will contain placeholders for the real addresses of the referenced libraries. These have to be updated via a process called linking before deploying the contract.

solcjs provides the `linkByteCode` method to link library addresses to the generated byte code.

Here is an example to demonstrate this:

```
var solc = require("solc");

var input = {
    "lib.sol": "library L { function f() returns (uint) { return 7; } }",
    "cont.sol": "import 'lib.sol'; contract x { function g() { L.f(); } }"
};

var output = solc.compile({sources: input}, 1);

var finalByteCode = solc.linkBytecode(output.contracts["x"].bytecode, {
'L': '0x123456...' });
```

## Updating the ABI

The ABI of a contract provides various kinds of information about the contract other than implementation. The ABI generated by two different versions of compilers may not match as higher versions support more solidity features than lower versions; therefore, they will include extra things in the ABI. For example, the fallback function was introduced in the 0.4.0 version of Solidity so the ABI generated using compilers whose version is less than 0.4.0 will have no information about fallback functions, and these smart contracts behave like they have a fallback function with an empty body and a payable modifier. So, the API should be updated so that applications that depend on the ABI of newer solidity versions can have better information about the contract.

solcjs provides an API to update the ABI. Here is an example code to demonstrate this:

```
var abi = require("solc/abi");

var inputABI =
[{"constant":false,"inputs":[],"name":"hello","outputs":[{"name":"","type":
"string"}],"payable":false,"type":"function"}];
var outputABI = abi.update("0.3.6", inputABI)
```

Here, 0.3.6 indicates that the ABI was generated using the 0.3.6 version of the compiler. As we are using solcjs version 0.4.8, the ABI will be updated to match the ABI generated by the 0.4.8 compiler version, not above it.

The output of the preceding code will be as follows:

```
[{"constant":false,"inputs":[],"name":"hello","outputs":[{"name":"","type":
"string"}],"payable":true,"type":"function"},{"type":"fallback","payable":t
rue}]
```

# Building a contract deployment platform

Now that we have learned how to use solcjs to compile solidity source code, it's time to build a platform that lets us write, compile, and deploy contracts. Our platform will let users provide their account address and private key, using which our platform will deploy contracts.

Before you start building the application, make sure that you are running the geth development instance, which is mining, has rpc enabled, and exposes eth, web3, and txpool APIs over the HTTP-RPC server. You can do all these by running this:

```
    geth --dev --rpc --rpccorsdomain "*" --rpcaddr "0.0.0.0" --rpcport
"8545" --mine --rpcapi "eth,txpool,web3"
```

# The project structure

In the exercise files of this chapter, you will find two directories, that is, Final and Initial. Final contains the final source code of the project, whereas Initial contains the empty source code files and libraries to get started with building the application quickly.

To test the Final directory, you will need to run npm install inside it and then run the app using the node app.js command inside the Final directory.

In the Initial directory, you will find a public directory and two files named app.js and package.json. The package.json file contains the backend dependencies on our app. app.js is where you will place the backend source code.

The public directory contains files related to the frontend. Inside public/css, you will find bootstrap.min.css, which is the bootstrap library, and you will also find the codemirror.css file, which is CSS of the codemirror library. Inside public/html, you will find index.html, where you will place the HTML code of our app and in the public/js directory, you will find .js files for codemirror and web3.js. Inside public/js, you will also find a main.js file, where you will place the frontend JS code of our app.

# Building the backend

Let's first build the backend of the app. First of all, run npm install inside the Initial directory to install the required dependencies for our backend.

Here is the backend code to run an express service and serve the `index.html` file and static files:

```
var express = require("express");
var app = express();

app.use(express.static("public"));

app.get("/", function(req, res){
 res.sendFile(__dirname + "/public/html/index.html");
})

app.listen(8080);
```

The preceding code is self-explanatory. Now let's proceed further. Our app will have two buttons, that is, **Compile** and **Deploy**. When the user clicks on the compile button, the contract will be compiled and when the deploy button is clicked on, the contract will be deployed.

We will be compiling and deploying contracts in the backend. Although this can be done in the frontend, we will do it in the backend because solcjs is available only for Node.js (although the JavaScript-based compilers it uses work on the frontend).

To learn how to compile on the frontend, go through the source code of solcjs, which will give you an idea about the APIs exposed by the JavaScript-based compiler.

When the user clicks on the compile button, the frontend will make a `GET` request to the `/compile` path by passing the contract source code. Here is the code for the route:

```
var solc = require("solc");

app.get("/compile", function(req, res){
 var output = solc.compile(req.query.code, 1);
 res.send(output);
})
```

At first, we import the solcjs library here. Then, we define the `/compile` route and inside the route callback, we simply compile the source code sent by the client with the optimizer enabled. And then we just send the `solc.compile` method's return value to the frontend and let the client check whether the compilation was successful or not.

When the user clicks on the deploy button, the frontend will make a `GET` request to the `/deploy` path by passing the contract source code and constructor arguments from the address and private key. When the user clicks on this button, the contract will be deployed and the transaction hash will be returned to the user.

Here is the code for this:

```
var Web3 = require("web3");
var BigNumber = require("bignumber.js");
var ethereumjsUtil = require("ethereumjs-util");
var ethereumjsTx = require("ethereumjs-tx");

var web3 = new Web3(new
Web3.providers.HttpProvider("http://localhost:8545"));

function etherSpentInPendingTransactions(address, callback)
{
 web3.currentProvider.sendAsync({
    method: "txpool_content",
    params: [],
    jsonrpc: "2.0",
    id: new Date().getTime()
 }, function (error, result) {
  if(result.result.pending)
  {
   if(result.result.pending[address])
   {
    var txns = result.result.pending[address];
    var cost = new BigNumber(0);

    for(var txn in txns)
    {
     cost = cost.add((new BigNumber(parseInt(txns[txn].value))).add((new
BigNumber(parseInt(txns[txn].gas))).mul(new
BigNumber(parseInt(txns[txn].gasPrice)))));
    }

    callback(null, web3.fromWei(cost, "ether"));
   }
   else
   {
    callback(null, "0");
   }
  }
  else
  {
   callback(null, "0");
  }
 })
}

function getNonce(address, callback)
{
 web3.eth.getTransactionCount(address, function(error, result){
  var txnsCount = result;

  web3.currentProvider.sendAsync({
     method: "txpool_content",
     params: [],
     jsonrpc: "2.0",
     id: new Date().getTime()
  }, function (error, result) {
   if(result.result.pending)
   {
    if(result.result.pending[address])
    {
     txnsCount = txnsCount +
Object.keys(result.result.pending[address]).length;
     callback(null, txnsCount);
    }
    else
    {
     callback(null, txnsCount);
    }
   }
   else
   {
    callback(null, txnsCount);
   }
  })
 })
}

app.get("/deploy", function(req, res){
 var code = req.query.code;
 var arguments = JSON.parse(req.query.arguments);
 var address = req.query.address;

 var output = solc.compile(code, 1);
 var contracts = output.contracts;

 for(var contractName in contracts)
 {
  var abi = JSON.parse(contracts[contractName].interface);
  var byteCode = contracts[contractName].bytecode;

  var contract = web3.eth.contract(abi);

  var data = contract.new.getData.call(null, ...arguments, {
   data: byteCode
  });

  var gasRequired = web3.eth.estimateGas({
      data: "0x" + data
  });

  web3.eth.getBalance(address, function(error, balance){
   var etherAvailable = web3.fromWei(balance, "ether");
   etherSpentInPendingTransactions(address, function(error, balance){
    etherAvailable = etherAvailable.sub(balance)
    if(etherAvailable.gte(web3.fromWei(new
BigNumber(web3.eth.gasPrice).mul(gasRequired), "ether")))
    {
     getNonce(address, function(error, nonce){
      var rawTx = {
             gasPrice: web3.toHex(web3.eth.gasPrice),
             gasLimit: web3.toHex(gasRequired),
             from: address,
             nonce: web3.toHex(nonce),
             data: "0x" + data
         };

      var privateKey = ethereumjsUtil.toBuffer(req.query.key, 'hex');
      var tx = new ethereumjsTx(rawTx);
      tx.sign(privateKey);

      web3.eth.sendRawTransaction("0x" + tx.serialize().toString('hex'),
function(err, hash) {
       res.send({result: {
        hash: hash,
       }});
      });
     })
    }
    else
    {
     res.send({error: "Insufficient Balance"});
    }
   })
  })

     break;
 }
})
```

This is how the preceding code works:

1. At first, the Web imports the `web3.js`, `BigNumber.js`, `ethereumjs-util`, and `ethereumjs-tx` libraries. Then, we create an instance of `Web3`.
2. Then, we define a function named `etherInSpentPendingTransactions`, which calculates the total ether that's being spent in the pending transactions of an address. As `web3.js` doesn't provide JavaScript APIs related to the transaction pool, we make a raw JSON-RPC call using `web3.currentProvider.sendAsync`. `sendAsync` is used to make raw JSON-RPC calls asynchronously. If you want to make this call synchronously, then use the `send` method instead of `sendAsync`. While calculating the total ether in the pending transactions of an address, we look for pending transactions in the transaction pool instead of the pending block due to the issue we discussed earlier. While calculating the total ether, we add the value and gas of each transaction as gas also deducted the ether balance.
3. Next, we define a function called `getNonce`, which retrieves the nonce of an address using the technique we discussed earlier. It simply adds the total number of mined transactions to the total number of pending transactions.
4. Finally, we declare the `/deploy` endpoint. At first, we compile the contract. Then, we deploy only the first contract. Our platform is designed to deploy the first contract if multiple contracts are found in the provided source code. You can later enhance the app to deploy all the compiled contracts instead of just the first one. Then, we create a contract object using `web3.eth.contract`.

5. As we aren't using the hooked-web3-provider or any hack to intercept `sendTransactions` and convert them into the `sendRawTransaction` call, in order to deploy the contract, we now need to generate the data part of the transaction, which will have the contract byte code and constructor arguments combined and encoded as a hexadecimal string. The contract object actually lets us generate the data of the transaction. This can be done by calling the `getData` method with function arguments. If you want to get data to deploy the contract, then call `contract.new.getData`, and if you want to call a function of the contract, then call `contract.functionName.getData`. In both the cases, provide the arguments to the `getData` method. So, in order to generate the data of a transaction, you just need the contract's ABI. To learn how the function name and arguments are combined and encoded to generate data, you can check out `https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI#examples`, but this won't be required if you have the ABI of the contract or know how to create the ABI manually.
6. Then, we use `web3.eth.estimateGas` to calculate the amount of gas that would be required to deploy the contract.
7. Later, we check whether the address has enough ether to pay for the gas required to deploy the contract. We find this out by retrieving the balance of the address and subtracting it with the balance spent in the pending transactions and then checking whether the remaining balance is greater than or equal to the amount of ether required for the gas.
8. And finally, we get the nonce, signing and broadcasting the transactions. We simply return the transaction hash to the frontend.

# Building the frontend

Now let's build the frontend of our application. Our frontend will contain an editor, using which the user writes code. And when the user clicks on the compile button, we will dynamically display input boxes where each input box will represent a constructor argument. When the deploy button is clicked on, the constructor argument values are taken from these input boxes. The user will need to enter the JSON string in these input boxes.

We will be using the codemirror library to integrate the editor in our frontend. To learn more about how to use codemirror, refer to `http://codemirror.net/`.

Here is the frontend HTML code of our app. Place this code in the `index.html` file:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
        <meta http-equiv="x-ua-compatible" content="ie=edge">
        <link rel="stylesheet" href="/css/bootstrap.min.css">
        <link rel="stylesheet" href="/css/codemirror.css">
        <style type="text/css">
            .CodeMirror
            {
                height: auto;
            }
        </style>
    </head>
    <body>
        <div class="container">
            <div class="row">
                <div class="col-md-6">
                    <br>
                    <textarea id="editor"></textarea>
                    <br>
                    <span id="errors"></span>
                    <button type="button" id="compile" class="btn btn-
primary">Compile</button>
                </div>
                <div class="col-md-6">
                    <br>
                    <form>
                        <div class="form-group">
                            <label for="address">Address</label>
                            <input type="text" class="form-control"
id="address" placeholder="Prefixed with 0x">
                        </div>
                        <div class="form-group">
                            <label for="key">Private Key</label>
                            <input type="text" class="form-control"
id="key" placeholder="Prefixed with 0x">
                        </div>
                        <hr>
                        <div id="arguments"></div>
                        <hr>
                        <button type="button" id="deploy" class="btn btn-
primary">Deploy</button>
                    </form>
                </div>
            </div>
        </div>
        <script src="/js/codemirror.js"></script>
        <script src="/js/main.js"></script>
    </body>
</html>
```

Here, you can see that we have a `textarea`. The `textarea` tag will hold whatever the user will enter in the codemirror editor. Everything else in the preceding code is self-explanatory.

Here is the complete frontend JavaScript code. Place this code in the `main.js` file:

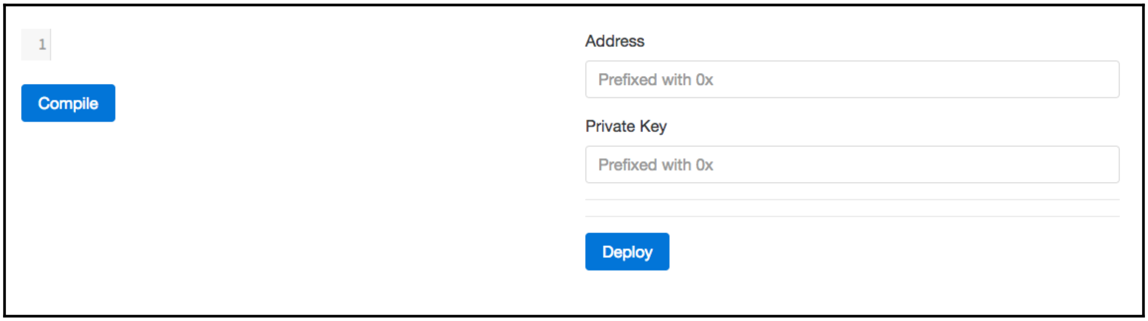
```
var editor = CodeMirror.fromTextArea(document.getElementById("editor"), {
    lineNumbers: true,
});

var argumentsCount = 0;

document.getElementById("compile").addEventListener("click", function(){
 editor.save();
 var xhttp = new XMLHttpRequest();

 xhttp.onreadystatechange = function() {
     if (this.readyState == 4 && this.status == 200) {
      if(JSON.parse(xhttp.responseText).errors != undefined)
      {
       document.getElementById("errors").innerHTML =
JSON.parse(xhttp.responseText).errors + "<br><br>";
      }
      else
      {
       document.getElementById("errors").innerHTML = "";
      }

      var contracts = JSON.parse(xhttp.responseText).contracts;

      for(var contractName in contracts)
      {
       var abi = JSON.parse(contracts[contractName].interface);

       document.getElementById("arguments").innerHTML = "";

       for(var count1 = 0; count1 < abi.length; count1++)
       {
        if(abi[count1].type == "constructor")
        {
         argumentsCount = abi[count1].inputs.length;

         document.getElementById("arguments").innerHTML =
'<label>Arguments</label>';

         for(var count2 = 0; count2 < abi[count1].inputs.length; count2++)
         {
          var inputElement = document.createElement("input");
          inputElement.setAttribute("type", "text");
          inputElement.setAttribute("class", "form-control");
          inputElement.setAttribute("placeholder",
abi[count1].inputs[count2].type);
          inputElement.setAttribute("id", "arguments-" + (count2 + 1));

          var br = document.createElement("br");

          document.getElementById("arguments").appendChild(br);
          document.getElementById("arguments").appendChild(inputElement);
         }

         break;
        }
       }

       break;
      }
     }
 };

 xhttp.open("GET", "/compile?code=" +
encodeURIComponent(document.getElementById("editor").value), true);
 xhttp.send();
})

document.getElementById("deploy").addEventListener("click", function(){
 editor.save();

 var arguments = [];

 for(var count = 1; count <= argumentsCount; count++)
 {
  arguments[count - 1] = JSON.parse(document.getElementById("arguments-" +
count).value);
 }

 var xhttp = new XMLHttpRequest();

 xhttp.onreadystatechange = function() {
     if (this.readyState == 4 && this.status == 200)
     {
      var res = JSON.parse(xhttp.responseText);

      if(res.error)
      {
       alert("Error: " + res.error)
      }
      else
      {
       alert("Txn Hash: " + res.result.hash);
      }
     }
     else if(this.readyState == 4)
     {
      alert("An error occured.");
     }
 };

 xhttp.open("GET", "/deploy?code=" +
encodeURIComponent(document.getElementById("editor").value) + "&arguments="
+ encodeURIComponent(JSON.stringify(arguments)) + "&address=" +
document.getElementById("address").value + "&key=" +
document.getElementById("key").value, true);
 xhttp.send();
})
```

Here is how the preceding code works:

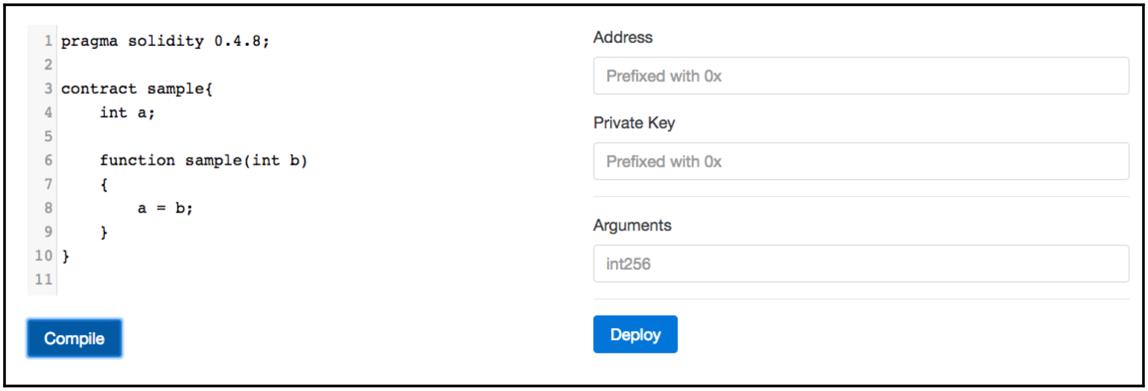
1. At first, we add the code editor to the web page. The code editor will be displayed in place of `textarea` and `textarea` will be hidden.
2. Then we have the compile button's click event handler. Inside it, we save the editor, which copies the content of the editor to `textarea`. When the compile button is clicked on, we make a request to the `/compile` path, and once we get the result, we parse it and display the input boxes so that the user can enter the constructor arguments. Here, we only read the constructor arguments for the first contract. But you can enhance the UI to display input boxes for constructors of all the contracts if there are more than one.
3. And finally, we have the deploy button's click event handler. Here, we read the constructor arguments' value, parsing and putting them in an array. And then we add a request to the `/deploy` endpoint by passing the address, key, code, and argument value. If there is an error, then we display that in a popup; otherwise, we display the transaction hash in the popup.

# Testing

To test the app, run the `app.js` node inside the `Initial` directory and visit `localhost:8080`. You will see what is shown in the following screenshot:

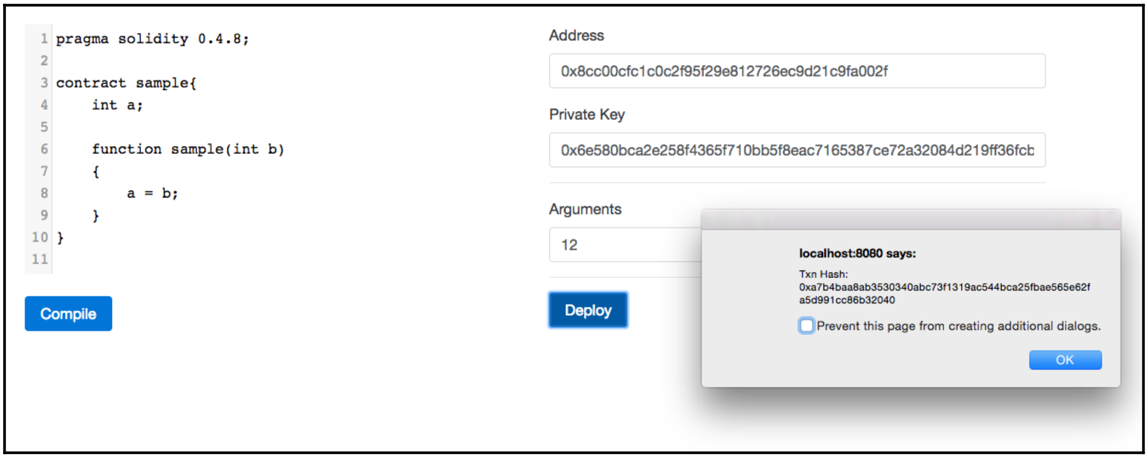


Now enter some solidity contract code and press the compile button. Then, you will be able to see new input boxes appearing on the right-hand side. For example, take a look at the following screenshot:



```
pragma solidity 0.4.8;

contract sample{
    int a;

    function sample(int b)
    {
        a = b;
    }
}

```

Now enter a valid address and its associated private key. And then enter values for the constructor arguments and click on deploy. If everything goes right, then you will see an alert box with the transaction hash. For example, take a look at the following screenshot:



```
pragma solidity 0.4.8;

contract sample{
    int a;

    function sample(int b)
    {
        a = b;
    }
}

```

Address

0x8cc00cfc1c0c2f95f29e812726ec9d21c9fa002f

Private Key

0x6e580bca2e258f4365f710bb5f8eac7165387ce72a32084d219ff36fcb

Arguments

12

**localhost:8080 says:**

Txn Hash:
0xa7b4baa8ab3530340abc73f1319ac544bca25fbae565e62f
a5d991cc86b32040

□ Prevent this page from creating additional dialogs.

# Summary

In this chapter, we learned how to use the transaction pool API, how to calculate a proper nonce, calculate a spendable balance, generate the data of a transaction, compile contracts, and so on. We then built a complete contract compilation and deployment platform. Now you can go ahead and enhance the application we have built to deploy all the contracts found in the editor, handle imports, add libraries, and so on.

In the next chapter, we will learn about Oraclize by building a decentralized betting app.

# 7

# Building a Betting App

Sometimes, it is necessary for smart contracts to access data from other dapps or from the World Wide Web. But it's really complicated to let smart contracts access outside data due to technical and consensus challenges. Therefore, currently, Ethereum smart contracts don't have native support to access outside data. But there are third-party solutions for Ethereum smart contracts to access data from some popular dapps and from the World Wide Web. In this chapter, we will learn how to use Oraclize to make HTTP requests from Ethereum smart contracts to access data from the World Wide Web. We will also learn how to access files stored in IPFS, use the strings library to work with strings, and so on. We will learn all this by building a football-betting smart contract and a client for it.

In this chapter, we'll cover the following topics:

- How does Oraclize work?
- What are Oraclize's various data sources and how do each of them work?
- How does consensus work in Oraclize?
- Integrating Oraclize in Ethereum smart contracts
- Using strings the `Solidity` library to make it easy to work with strings
- Building a football betting app

## Introduction to Oraclize

Oraclize is a service that aims to enable smart contracts to access data from other blockchains and the World Wide Web. This service is currently live on bitcoin and Ethereum's testnet and mainnet. What makes Oraclize so special is that you don't need to trust it because it provides proof of authenticity of all data it provides to smart contracts.

In this chapter, our aim is to learn how Ethereum smart contracts can use the Oraclize service to fetch data from the World Wide Web.

# How does it work?

Let's look at the process by which an Ethereum smart contract can fetch data from other blockchains and the World Wide Web using Oraclize.

To fetch data that exists outside of the Ethereum blockchain, an Ethereum smart contract needs to send a query to Oraclize, mentioning the data source (representing where to fetch the data from) and the input for the data source (representing what to fetch).

Sending a query to Oraclize Oraclize means sending a contract call (that is, an internal transaction) to the Oraclize contract present in the Ethereum blockchain.

The Oraclize server keeps looking for new incoming queries to its smart contract. Whenever it sees a new query, it fetches the result and sends it back to your contract by calling the `_callback` method of your contract.

# Data sources

Here is a list of sources from which Oraclize lets smart contracts fetch data:

- URL: The URL data source provides you with the ability to make an HTTP GET or POST request, that is, fetch data from the WWW.
- `WolframAlpha`: The `WolframAlpha` data source provides you with the ability to submit a query to the `WolframAlpha` knowledge engine and get the answer.
- `Blockchain`: The `blockchain` data source provides you with ability to access data from other `blockchains`. Possible queries that can be submitted to the `blockchain` data source are `bitcoin blockchain height`, `litecoin hashrate`, `bitcoin difficulty`, `1NPFRDJuEdyqEn2nmLNaWMfojNksFjbL4S balance`, and so on.
- `IPFS`: The `IPFS` data source provides you with the ability to fetch the content of a file stored in `IPFS`.

- `Nested`: The `nested` data source is a metadata source; it does not provide access to additional services. It was designed to provide a simple aggregation logic, enabling a single query to leverage sub-queries based on any available data source and producing a single string as a result; for example:

  ```
  [WolframAlpha] temperature in ${[IPFS]
  QmP2ZkdsJG7LTw7jBbizTTgY1ZBeen64PqMgCAWz2koJBL}.
  ```

- `Computation`: The `computation` data source enables the auditable execution of a given application into a secure off-chain context; that is, it lets us fetch the result of an off-chain execution of an application. This application has to print the query result on the last line (on the standard output) before its quits. The execution context has to be described by a Dockerfile, where building and running it should start your main application straight away. The Dockerfile initialization plus your application execution should terminate as soon as possible: the maximum execution timeout is 5 minutes on an `AWS t2.micro` instance. Here, we are considering an `AWS t2.micro` instance because that's what Oraclize will use to execute the application. As the input for the data source is the IPFS multihash of a ZIP archive containing such files (Dockerfile plus any external file dependencies, and the Dockerfile has to be placed in the archive root), you should take care of preparing this archive and pushing it to IPFS beforehand.

These data sources are available at the time of writing this book. But many more data sources are likely to be available in the future.

# Proof of authenticity

Although Oraclize is a trusted service, you may still want to check whether the data returned by Oraclize is authentic or not, that is, whether it was manipulated by Oraclize or someone else in transit.

Optionally, Oraclize provides the TLSNotary proof of result that's returned from the URL, blockchain, and nested and computation data sources. This proof is not available for `WolframAlpha` and `IPFS` data sources. Currently, Oraclize only supports the TLSNotary proof, but in the future, they may support some other ways to authenticate. Currently, the TLSNotary proof needs to be validated manually, but Oraclize is already working on on-chain proof verification; that is, your smart contract code can verify the TLSNotary proof on its own while receiving the data from Oraclize so that this data is discarded if the proof turns out to be invalid.

This tool (`https://github.com/Oraclize/proof-verification-tool`) is an open source tool provided by Oraclize to validate the TLSNotary proof in case you want to.

Understanding how TLSNotary works is not required to use Oraclize or to verify the proof. The tool to validate the TLSNotary proof is open source; therefore, if it contains any malicious code, then it can easily be caught, so this tool can be trusted.

Let's look at a high-level overview of how TLSNotary works. To understand how TLSNotary works, you need to first understand how TLS works. The TLS protocol provides a way for the client and server to create an encrypted session so that no one else can read or manipulate what is transferred between the client and server. The server first sends its certificate (issued to the domain owner by a trusted CA) to the client. The certificate will contain the public key of the server. The client uses the CA's public key to decrypt the certificate so that it can verify that the certificate is actually issued by the CA and get the server's public key. Then, the client generates a symmetric encryption key and a MAC key and encrypts them using the server's public key and sends it to the server. The server can only decrypt this message as it has the private key to decrypt it. Now the client and server share the same symmetric and MAC keys and no one else knows about these keys and they can start sending and receiving data from each other. The symmetric key is used to encrypt and decrypt the data where the MAC key and the symmetric key together are used to generate a signature for the encrypted message so that in case the message is modified by an attacker, the other party can know about it.

TLSNotary is a modification of TLS, which is used by Oraclize to provide cryptography proof showing that the data they provided to your smart contract was really the one the data source gave to Oraclize at a specific time. Actually the TLSNotary protocol is an open source technology, developed and used by the PageSigner project.

TLSNotary works by splitting the symmetric key and the MAC key among three parties, that is, the server, an auditee, and an auditor. The basic idea of TLSNotary is that the auditee can prove to the auditor that a particular result was returned by the server at a given time.

So here is an overview of how exactly TLSNotary lets us achieve this. The auditor calculates the symmetric key and MAC key and gives only the symmetric key to the auditee. The MAC key is not needed by the auditee as the MAC signature check ensures that the TLS data from the server was not modified in transit. With the symmetric encryption key, the auditee can now decrypt data from the server. Because all messages are "signed" by the bank using the MAC key and only the server and the auditor know the MAC key, a correct MAC signature can serve as proof that certain messages came from the bank and were not spoofed by the auditee.

In the case of the Oraclize service, Oraclize is the auditee, while a locked-down AWS instance of a specially designed, open source Amazon machine image acts as the auditor.

The proof data they provide are the signed attestations of this AWS instance that a proper TLSnotary proof did occur. They also provide some additional proof regarding the software running in the AWS instance, that is, whether it has been modified since being initialized.

# Pricing

The first Oraclize query call coming from any Ethereum address is completely free of charge. Oraclize calls are free when used on testnets! This works for moderate usage in test environments only.

From the second call onward, you have to pay in ether for queries. While sending a query to Oraclize (that is, while making an internal transaction call), a fee is deducted by transferring ether from the calling contract to the Oraclize contract. The amount of ether to deduct depends on the data source and proof type.

Here is a table that shows the number of ether that is deducted while sending a query:

| **Data source** | **Without proof** | **With TLSNotary proof** |
|---|---|---|
| URL | $0.01 | $0.05 |
| Blockchain | $0.01 | $0.05 |
| WolframAlpha | $0.03 | $0.03 |
| IPFS | $0.01 | $0.01 |

So if you are making a HTTP request and you want the TLSNotary proof too, then the calling contract must have an ether worth of $0.05; otherwise, an exception is thrown.

# Getting started with the Oraclize API

For a contract to use the Oraclize service, it needs to inherit the `usingOraclize` contract. You can find this contract at `https://github.com/Oraclize/Ethereum-api`.

The usingOraclize contract acts as the proxy for the OraclizeI and OraclizeAddrResolverI contracts. Actually, usingOraclize makes it easy to make calls to the OraclizeI and OraclizeAddrResolverI contracts, that is, it provides simpler APIs. You can also directly make calls to the OraclizeI and OraclizeAddrResolverI contracts if you feel comfortable. You can go through the source code of these contracts to find all the available APIs. We will only learn the most necessary ones.

Let's look at how to set proof type, set proof storage location, make queries, find the cost of a query, and so on.

## Setting the proof type and storage location

Whether you need the TLSNotary proof from Oraclize or not, you have to specify the proof type and proof storage location before making queries.

If you don't want proof, then put this code in your contract:

```
oraclize_setProof(proofType_NONE)
```

And if you want proof, then put this code in your contract:

```
oraclize_setProof(proofType_TLSNotary | proofStorage_IPFS)
```

Currently, proofStorage_IPFS is the only proof storage location available; that is, TLSNotary proof is only stored in IPFS.

You may execute any of these methods just once, for instance, in the constructor or at any other time if, for instance, you need the proof for certain queries only.

## Sending queries

To send a query to Oraclize, you will need to call the oraclize_query function. This function expects at least two arguments, that is, the data source and the input for the given data source. The data source argument is not case-sensitive.

Here are some basic examples of the oraclize_query function:

```
oraclize_query("WolframAlpha", "random number between 0 and 100");

oraclize_query("URL",
"https://api.kraken.com/0/public/Ticker?pair=ETHXBT");

oraclize_query("IPFS", "QmdEJwJG1T9rzHvBD8i69HHuJaRgXRKEQCP7Bh1BVttZbU");
oraclize_query("URL", "https://xyz.io/makePayment", '{"currency": "USD",
"amount": "1"}');
```

Here is how the preceding code works:

- If the first argument is a string, it is assumed to be the data source and the second argument is assumed to be the input for the data source. In the first call, the data source is `WolframAlpha` and the search query we sent to it was `random number between 0 and 100`.
- In the second call, we make an `HTTP GET` request to the `URL` present in the second argument.
- In the third call, we fetch the content of the `QmdEJwJG1T9rzHvBD8i69HHuJaRgXRKEQCP7Bh1BVttZbU` file from `IPFS`.
- If two consecutive arguments after the data source are strings, then it's assumed to be a POST request. In the last call, we make an `HTTP POST` request to `https://xyz.io/makePayment` and the `POST` request body content is the string in the third argument. Oraclize is intelligent enough to detect the content-type header based on the string format.

## Scheduling queries

If you want Oraclize to execute your query at a scheduled future time, just specify the delay (in seconds) from the current time as the first argument.

Here is an example:

```
oraclize_query(60, "WolframAlpha", "random number between 0 and 100");
```

The preceding query will be executed by Oraclize `60` seconds after it's been seen. So if the first argument is a number, then it's assumed that we are scheduling a query.

## Custom gas

The transaction originating from Oraclize to your `__callback` function costs gas, just like any other transaction. You need to pay Oraclize the gas cost. The ether oraclize_query charges to make a query are also used to provide gas while calling the `__callback` function. By default, Oraclize provides 200,000 gas while calling the `__callback` function.

This return gas cost is actually in your control since you write the code in the `__callback` method and as such, can estimate it. So, when placing a query with Oraclize, you can also specify how much the `gasLimit` should be on the `__callback` transaction. Note, however, that since Oraclize sends the transaction, any unspent gas is returned to Oraclize, not you.

If the default, and minimum, value of 200,000 gas is not enough, you can increase it by specifying a different `gasLimit` in this way:

```
oraclize_query("WolframAlpha", "random number between 0 and 100", 500000);
```

Here, you can see that if the last argument is a number, then it's assumed to be the custom gas. In the preceding code, Oraclize will use a 500k `gasLimit` for the callback transaction instead of 200k. Because we are asking Oraclize to provide more gas, Oraclize will deduct more ether (depending on how much gas is required) while calling `oraclize_query`.

> Note that if you offer too low a `gasLimit`, and your `__callback` method is long, you may never see a callback. Also note that the custom gas has to be more than 200k.

## Callback functions

Once your result is ready, Oraclize will send a transaction back to your contract address and invoke one of these three methods:

- either `__callback(bytes32 myid, string result).Myid` is a unique ID for every query. This ID is returned by the `oraclize_query` method. If you have multiple `oraclize_query` calls in your contract, then this is used to match the query this result is for.
- If you requested for the TLS Notary proof, this is the result: `__callback(bytes32 myid, string result, bytes proof)`
- As a last resort, if the other methods are absent, the fallback function is `function()`

Here is an example of the `__callback` function:

```
function __callback(bytes32 myid, string result) {
    if (msg.sender != oraclize_cbAddress()) throw; // just to be sure the
calling address is the Oraclize authorized one

    //now doing something with the result..
}
```

## Parsing helpers

The result returned from an HTTP request can be HTML, JSON, XML, binary, and so on. In Solidity, it is difficult and expensive to parse the result. Due to this, Oraclize provides parsing helpers to let it handle the parsing on its servers, and you get only the part of the result that you need.

To ask Oraclize to parse the result, you need to wrap the URL with one of these parsing helpers:

- `xml(..)` and `json(..)` helpers let you ask Oraclize to only return part of the JSON or XML-parsed response; for example, take a look at the following:
    - In order to get the whole response back, you use the `URL` data source with the `api.kraken.com/0/public/Ticker?pair=ETHUSD` URL argument
    - If all you want is the last-price field, you need to use the JSON parsing call as `json(api.kraken.com/0/public/Ticker?pair=ETHUSD).result.XETHZUSD.c.0`
- The `html(..).xpath(..)` helper is useful for HTML scraping. Just specify the XPATH you want as the `xpath(..)` argument; for example, take a look at the following:
    - To fetch the text of a specific tweet, use `html(https://twitter.com/oraclizeit/status/671316655893561344).xpath(//*[contains(@class, 'tweet-text')]/text())`.
- The `binary(..)` helper is useful in getting binary files such as certificate files:
    - To fetch only a portion of the binary file, you can use `slice(offset,length)`; the first parameter is the offset, while the second one is the length of the slice you want back (both in bytes).
    - Example: Fetch only the first 300 bytes from a binary CRL, `binary(https://www.sk.ee/crls/esteid/esteid2015.crl).slice(0,300)`. The binary helper must be used with the slice option, and only binary files (not encoded) are accepted.

If and when the server is not responding or is unreachable, we will send you an empty response. You can test queries using `http://app.Oraclize.it/home/test_query`.

## Getting the query price

If you would like to know how much a query would cost before making the actual query, then you can use the `Oraclize.getPrice()` function to get the amount of wei required. The first argument it takes is the data source, and the second argument is optional, which is the custom gas.

One popular use case of this is to notify the client to add ether to the contract if there isn't enough to make the query.

# Encrypting queries

Sometimes, you may not want to reveal the data source and/or the input for the data source. For example: you may not want to reveal the API key in the URL if there is any. Therefore, Oraclize provides a way to store queries encrypted in the smart contract and only Oraclize's server has the key to decrypt it.

Oraclize provides a Python tool (`https://github.com/Oraclize/encrypted-queries`), which can be used to encrypt the data source and/or the data input. It generates a non-deterministic encrypted string.

The CLI command to encrypt an arbitrary string of text is as follows:

```
    python encrypted_queries_tools.py -e -p
044992e9473b7d90ca54d2886c7addd14a61109af202f1c95e218b0c99eb060c7134c4ae463
45d0383ac996185762f04997d6fd6c393c86e4325c469741e64eca9 "YOUR DATASOURCE or
INPUT"
```

The long hexadecimal string you see is the public key of Oraclize's server. Now you can use the output of the preceding command in place of the data source and/or the input for the data source.

> In order to prevent the misuse of encrypted queries (that is, replay attacks) the first contract querying Oraclize with a specific encrypted query becomes its rightful owner. Any other contract reusing the exact same string will not be allowed to use it and will receive an empty result. As a consequence, remember to always generate a newly encrypted string when redeploying contracts using encrypted queries.

## Decrypting the data source

There is another data source called decrypt. It is used to decrypt an encrypted string. But this data source doesn't return any result; otherwise, anyone would have the ability to decrypt the data source and input for the data source.

It was specifically designed to be used within the nested data source to enable partial query encryption. It is its only use case.

# Oraclize web IDE

Oraclize provides a web IDE, using which you can write, compile, and test Oraclize-based applications. You can find it at `http://dapps.Oraclize.it/browser-Solidity/`.

If you visit the link, then you will notice that it looks exactly the same as browser Solidity. And it's actually browser Solidity with one extra feature. To understand what that feature is, we need to understand browser Solidity more in depth.

Browser Solidity not only lets us write, compile, and generate web3.js code for our contracts, but it also lets us test those contracts there itself. Until now, in order to test our contract, we were setting up an Ethereum node and sending transactions to it. But browser Solidity can execute contracts without connecting to any node and everything happens in memory. It achieves this using ethereumjs-vm, which is a JavaScript implementation of EVM. Using ethereumjs-vm, you can create our own EVM and run byte code. If we want, we can configure browser Solidity to use the Ethereum node by providing the URL to connect to. The UI is very informative; therefore, you can try all these by yourself.

What's special about the Oraclize web IDE is that it deploys the Oraclize contract in the in-memory execution environment so that you don't have to connect to the testnet or mainnet node, but if you use browser Solidity, then you have to connect to the testnet or mainnet node to test Oraclize APIs.

You can find more resources related to integrating Oraclize at `https://dev.Oraclize.it/`.

# Working with strings

Working with strings in Solidity is not as easy as working with strings in other high-level programming languages, such as JavaScript, Python, and so on. Therefore, many Solidity programmers have come up with various libraries and contracts to make it easy to work with strings.

The `strings` library is the most popular strings utility library. It lets us join, concatenate, split, compare, and so on by converting a string to something called a slice. A slice is a struct that holds the length of the string and the address of the string. Since a slice only has to specify an offset and a length, copying and manipulating slices is a lot less expensive than copying and manipulating the strings they reference.

To further reduce gas costs, most functions on slice that need to return a slice modify the original one instead of allocating a new one; for instance, `s.split(".")` will return the text up to the first `"."`, modifying s to only contain the remainder of the string after the `"."`. In situations where you do not want to modify the original slice, you can make a copy with `.copy()`, for example, `s.copy().split(".")`. Try and avoid using this idiom in loops; since Solidity has no memory management, it will result in allocating many short-lived slices that are later discarded.

Functions that have to copy string data will return strings rather than slices; these can be cast back to slices for further processing if required.

Let's look at a few examples of working with strings using the `strings` library:

```
pragma Solidity ^0.4.0;

import "github.com/Arachnid/Solidity-stringutils/strings.sol";

contract Contract
{
    using strings for *;

    function Contract()
    {
        //convert string to slice
        var slice = "xyz abc".toSlice();

        //length of string
        var length = slice.len();

        //split a string
        //subslice = xyz
        //slice = abc
```

```
            var subslice = slice.split(" ".toSlice());

            //split a string into an array
            var s = "www.google.com".toSlice();
            var delim = ".".toSlice();
            var parts = new string[](s.count(delim));
            for(uint i = 0; i < parts.length; i++) {
                parts[i] = s.split(delim).toString();
            }

            //Converting a slice back to a string
            var myString = slice.toString();

            //Concatenating strings
            var finalSlice = subslice.concat(slice);

            //check if two strings are equal
            if(slice.equals(subslice))
            {

            }
        }
    }
```

The preceding code is self-explanatory.

Functions that return two slices come in two versions: a nonallocating version that takes the second slice as an argument, modifying it in place, and an allocating version that allocates and returns the second slice; for example, let's take a look at the following:

```
    var slice1 = "abc".toSlice();

    //moves the string pointer of slice1 to point to the next rune (letter)
    //and returns a slice containing only the first rune
    var slice2 = slice1.nextRune();

    var slice3 = "abc".toSlice();
    var slice4 = "".toSlice();

    //Extracts the first rune from slice3 into slice4, advancing the slice to
    point to the next rune and returns slice4.
    var slice5 = slice3.nextRune(slice4);
```

You can learn more about the strings library at `https://github.com/Arachnid/Solidity-stringutils`.

# Building the betting contract

In our betting application, two people can choose to bet on a football match with one person supporting the home team and the other person supporting the away team. They both should bet the same amount of money, and the winner takes all the money. If the match is a draw, then they both will take back their money.

We will use the FastestLiveScores API to find out the result of matches. It provides a free API, which lets us make 100 requests per hour for free. First, go ahead and create an account and then generate an API key. To create an account, visit `https://customer.fastestlivescores.com/register`, and once the account is created, you will have the API key visible at `https://customer.fastestlivescores.com/`. You can find the API documentation at `https://docs.crowdscores.com/`.

For every bet between two people in our application, a betting contract will be deployed. The contract will contain the match ID retrieved from the `FastestLiveScores` API, the amount of wei each of the parties need to invest, and the addresses of the parties. Once both parties have invested in the contract, they will find out the result of the match. If the match is not yet finished, then they will try to check the result after every 24 hours.

Here is the code for the contract:

```
pragma Solidity ^0.4.0;

import "github.com/Oraclize/Ethereum-api/oraclizeAPI.sol";
import "github.com/Arachnid/Solidity-stringutils/strings.sol";

contract Betting is usingOraclize
{
    using strings for *;

    string public matchId;
    uint public amount;
    string public url;

    address public homeBet;
    address public awayBet;

    function Betting(string _matchId, uint _amount, string _url)
    {
        matchId = _matchId;
        amount = _amount;
        url = _url;

        oraclize_setProof(proofType_TLSNotary | proofStorage_IPFS);
    }

    //1 indicates home team
    //2 indicates away team
    function betOnTeam(uint team) payable
    {

        if(team == 1)
        {
            if(homeBet == 0)
            {
                if(msg.value == amount)
                {
                    homeBet = msg.sender;
                    if(homeBet != 0 && awayBet != 0)
                    {
                        oraclize_query("URL", url);
                    }
                }
                else
                {
                    throw;
                }
            }
            else
            {
                throw;
            }
        }
        else if(team == 2)
        {
            if(awayBet == 0)
            {
                if(msg.value == amount)
                {
                    awayBet = msg.sender;

                    if(homeBet != 0 && awayBet != 0)
                    {
                        oraclize_query("URL", url);
                    }
                }
                else
                {
                    throw;
                }
            }
            else
            {
                throw;
            }
        }
        else
        {
            throw;
        }
    }

    function __callback(bytes32 myid, string result, bytes proof) {
        if (msg.sender != oraclize_cbAddress())
        {
            throw;
        }
        else
        {
            if(result.toSlice().equals("home".toSlice()))
            {
                homeBet.send(this.balance);
            }
            else if(result.toSlice().equals("away".toSlice()))
            {
                awayBet.send(this.balance);
            }
            else if(result.toSlice().equals("draw".toSlice()))
            {
                homeBet.send(this.balance / 2);
                awayBet.send(this.balance / 2);
            }
            else
            {
                if (Oraclize.getPrice("URL") < this.balance)
                {
                    oraclize_query(86400, "URL", url);
                }
            }
        }
    }
}
```

The contract code is self-explanatory. Now compile the preceding code using `solc.js` or browser Solidity depending on whatever you are comfortable with. You will not need to link the `strings` library because all the functions in it are set to the `internal` visibility.

In browser Solidity, when specifying to import a library or contract from the HTTP URL, make sure that it's hosted on GitHub; otherwise, it won't fetch it. In that GitHub file URL, make sure that you remove the protocol as well as `blob/{branch-name}`.

# Building a client for the betting contract

To make it easy to find match's IDs, deploy, and invest in contracts, we need to build a UI client. So let's get started with building a client, which will have two paths, that is, the home path to deploy contracts and bet on matches and the other path to find the list of matches. We will let users deploy and bet using their own offline accounts so that the entire process of betting happens in a decentralized manner and nobody can cheat.

Before we start building our client, make sure that you have testnet synced because Oraclize works only on Ethereum's testnet/mainnet and not on private networks. You can switch to testnet and start downloading the testnet blockchain by replacing the `--dev` option with the `--testnet` option. For example, take a look at the following:

```
geth --testnet --rpc --rpccorsdomain "*" --rpcaddr "0.0.0.0" --rpcport
"8545"
```

## Projecting the structure

In the exercise files of this chapter, you will find two directories, that is, Final and Initial. Final contains the final source code of the project, whereas Initial contains the empty source code files and libraries to get started with building the application quickly.

To test the `Final` directory, you will need to run `npm install` inside it and then run the app using the `node app.js` command inside the Final directory.

In the `Initial` directory, you will find a `public` directory and two files named `app.js` and `package.json`. The `package.json` file contains the backend dependencies of our app, and `app.js` is where you will place the backend source code.

The public directory contains files related to the frontend. Inside public/css, you will find bootstrap.min.css, which is the bootstrap library. Inside public/html, you will find the index.html and matches.ejs files, where you will place the HTML code of our app, and in the public/js directory, you will find js files for web3.js, and ethereumjs-tx. Inside public/js, you will also find a main.js file, where you will place the frontend JS code of our app. You will also find the Oraclize Python tool to encrypt queries.

## Building the backend

Let's first build the backend of the app. First of all, run npm install inside the Initial directory to install the required dependencies for our backend.

Here is the backend code to run an express service and serve the index.html file and static files and set the view engine:

```
var express = require("express");
var app = express();

app.set("view engine", "ejs");

app.use(express.static("public"));

app.listen(8080);

app.get("/", function(req, res) {
    res.sendFile(__dirname + "/public/html/index.html");
})
```

The preceding code is self-explanatory. Now let's proceed further. Our app will have another page, which will display a list of recent matches with matches' IDs and result if a match has finished. Here is the code for the endpoint:

```
var request = require("request");
var moment = require("moment");

app.get("/matches", function(req, res) {
request("https://api.crowdscores.com/v1/matches?api_key=7b7a988932de4eaab4e
d1b4dcdc1a82a", function(error, response, body) {
        if (!error && response.statusCode == 200) {
            body = JSON.parse(body);

            for (var i = 0; i < body.length; i++) {
             body[i].start = moment.unix(body[i].start /
               1000).format("YYYY MMM DD hh:mm:ss");
            }
            res.render(__dirname + "/public/html/matches.ejs", {
                matches: body
            });
        } else {
            res.send("An error occured");
        }
    })
})
```

Here, we are making the API request to fetch the list of recent matches and then we are passing the result to the `matches.ejs` file so that it can render the result in a user-friendly UI. The API results give us the match start time as a timestamp; therefore, we are using moment to convert it to a human readable format. We make this request from the backend and not from the frontend so that we don't expose the API key to the users.

Our backend will provide an API to the frontend, using which the frontend can encrypt the query before deploying the contract. Our application will not prompt users to create an API key, as it would be a bad UX practice. The application's developer controlling the API key will cause no harm as the developer cannot modify the result from the API servers; therefore, users will still trust the app even after the application's developer knows the API key.

Here is code for the encryption endpoint:

```
var PythonShell = require("python-shell");

app.get("/getURL", function(req, res) {
    var matchId = req.query.matchId;

    var options = {
        args: ["-e", "-p",
"044992e9473b7d90ca54d2886c7addd14a61109af202f1c95e218b0c99eb060c7134c4ae46
345d0383ac996185762f04997d6fd6c393c86e4325c469741e64eca9",
"json(https://api.crowdscores.com/v1/matches/" + matchId +
"?api_key=7b7a988932de4eaab4ed1b4dcdc1a82a).outcome.winner"],
        scriptPath: __dirname
    };

    PythonShell.run("encrypted_queries_tools.py", options, function
      (err, results) {
        if(err)
        {
            res.send("An error occured");
        }
        else
        {
            res.send(results[0]);
            }
        });
    })
```

We have already seen how to use this tool. To run this endpoint successfully, make sure that Python is installed on your system. Even if Python is installed, this endpoint may show errors, indicating that Python's cryptography and base58 modules aren't installed. So make sure you install these modules if the tool prompts you to.

# Building the frontend

Now let's build the frontend of our application. Our frontend will let users see the list of recent matches, deploy the betting contract, bet on a game, and let them see information about a betting contract.

Let's first implement the `matches.ejs` file, which will display the list of recent matches. Here is the code for this:

```
    <!DOCTYPE html>
    <html lang="en">
        <head>
             <meta charset="utf-8">
             <meta name="viewport" content="width=device-width, initial-
    scale=1, shrink-to-fit=no">
             <meta http-equiv="x-ua-compatible" content="ie=edge">
             <link rel="stylesheet" href="/css/bootstrap.min.css">
         </head>
         <body>
             <div class="container">
                 <br>
                 <div class="row m-t-1">
                     <div class="col-md-12">
                         <a href="/">Home</a>
                     </div>
                 </div>
                 <br>
                 <div class="row">
                     <div class="col-md-12">
                         <table class="table table-inverse">
                               <thead>
                                 <tr>
                                     <th>Match ID</th>
                                     <th>Start Time</th>
                                     <th>Home Team</th>
                                     <th>Away Team</th>
                                     <th>Winner</th>
                                 </tr>
                               </thead>
                               <tbody>
                                   <% for(var i=0; i < matches.length; i++) {
    %>
                                       <tr>
                                           <td><%= matches[i].dbid %></td>
                                           <% if (matches[i].start) { %>
                                            <td><%= matches[i].start %></td>
                                         <% } else { %>
                                             <td>Time not finalized</td>
                                         <% } %>
                                           <td><%= matches[i].homeTeam.name
    %></td>
                                           <td><%= matches[i].awayTeam.name
    %></td>
                                           <% if (matches[i].outcome) { %>
                                            <td><%= matches[i].outcome.winner
    %></td>
                                         <% } else { %>
                                             <td>Match not finished</td>
                                         <% } %>
                                     </tr>
                                 <% } %>
                               </tbody>
                         </table>
                     </div>
                 </div>
             </div>
         </body>
     </html>
```

The preceding code is self-explanatory. Now let's write the HTML code for our home page. Our home page will display three forms. The first form is to deploy a betting contract, the second form is to invest in a betting contract, and the third form is to display information on a deployed betting contract.

Here is the HTML code for the home page. Place this code in the `index.html` page:

```
<!DOCTYPE html>
 <html lang="en">
     <head>
         <meta charset="utf-8">
         <meta name="viewport" content="width=device-width, initial-
scale=1, shrink-to-fit=no">
         <meta http-equiv="x-ua-compatible" content="ie=edge">
         <link rel="stylesheet" href="/css/bootstrap.min.css">
     </head>
     <body>
         <div class="container">
             <br>
             <div class="row m-t-1">
                 <div class="col-md-12">
                     <a href="/matches">Matches</a>
                 </div>
             </div>
             <br>
             <div class="row">
                 <div class="col-md-4">
                     <h3>Deploy betting contract</h3>
                     <form id="deploy">
                         <div class="form-group">
                             <label>From address: </label>
                             <input type="text" class="form-control"
id="fromAddress">
                         </div>
                         <div class="form-group">
                             <label>Private Key: </label>
                             <input type="text" class="form-control"
id="privateKey">
                         </div>
                         <div class="form-group">
                             <label>Match ID: </label>
                             <input type="text" class="form-control"
id="matchId">
                         </div>
                         <div class="form-group">
                             <label>Bet Amount (in ether): </label>
                             <input type="text" class="form-control"
id="betAmount">
                         </div>
                         <p id="message" style="word-wrap: break-word"></p>
                         <input type="submit" value="Deploy" class="btn
btn-primary" />
                     </form>
                 </div>
                 <div class="col-md-4">
                     <h3>Bet on a contract</h3>
                     <form id="bet">
                         <div class="form-group">
                             <label>From address: </label>
                             <input type="text" class="form-control"
id="fromAddress">
                         </div>
                         <div class="form-group">
```

```
                        <label>Private Key: </label>
                        <input type="text" class="form-control"
id="privateKey">
                    </div>
                    <div class="form-group">
                        <label>Contract Address: </label>
                        <input type="text" class="form-control"
id="contractAddress">
                    </div>
                    <div class="form-group">
                        <label>Team: </label>
                        <select class="form-control" id="team">
                            <option>Home</option>
                            <option>Away</option>
                        </select>
                    </div>
                    <p id="message" style="word-wrap: break-word"></p>
                    <input type="submit" value="Bet" class="btn btn-
primary" />
                </form>
            </div>
            <div class="col-md-4">
                <h3>Display betting contract</h3>
                <form id="find">
                    <div class="form-group">
                        <label>Contract Address: </label>
                        <input type="text" class="form-control"
 d="contractAddress">
                    </div>
                    <p id="message"></p>
                    <input type="submit" value="Find" class="btn btn-
primary" />
                </form>
            </div>
        </div>
    </div>

    <script type="text/javascript" src="/js/web3.min.js"></script>
    <script type="text/javascript" src="/js/ethereumjs-
tx.js"></script>
    <script type="text/javascript" src="/js/main.js"></script>
  </body>
</html>
```

The preceding code is self-explanatory. Now let's write JavaScript code to actually deploy the contract, invest in contracts, and display information on contracts. Here is the code for all this. Place this code in the `main.js` file:

```
var bettingContractByteCode = "6060604...";
var bettingContractABI =
[{"constant":false,"inputs":[{"name":"team","type":"uint256"}],"name":"betO
nTeam","outputs":[],"payable":true,"type":"function"},{"constant":false,"in
puts":[{"name":"myid","type":"bytes32"},{"name":"result","type":"string"}],
"name":"__callback","outputs":[],"payable":false,"type":"function"},{"const
ant":false,"inputs":[{"name":"myid","type":"bytes32"},{"name":"result","typ
e":"string"},{"name":"proof","type":"bytes"}],"name":"__callback","outputs"
:[],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":
"url","outputs":[{"name":"","type":"string"}],"payable":false,"type":"funct
ion"},{"constant":true,"inputs":[],"name":"matchId","outputs":[{"name":"","
type":"string"}],"payable":false,"type":"function"},{"constant":true,"input
s":[],"name":"amount","outputs":[{"name":"","type":"uint256"}],"payable":fa
lse,"type":"function"},{"constant":true,"inputs":[],"name":"homeBet","outpu
ts":[{"name":"","type":"address"}],"payable":false,"type":"function"},{"con
stant":true,"inputs":[],"name":"awayBet","outputs":[{"name":"","type":"addr
ess"}],"payable":false,"type":"function"},{"inputs":[{"name":"_matchId","ty
pe":"string"},{"name":"_amount","type":"uint256"},{"name":"_url","type":"st
ring"}],"payable":false,"type":"constructor"}];

var web3 = new Web3(new
Web3.providers.HttpProvider("http://localhost:8545"));

function getAJAXObject()
{
   var request;
   if (window.XMLHttpRequest) {
       request = new XMLHttpRequest();
   } else if (window.ActiveXObject) {
       try {
           request = new ActiveXObject("Msxml2.XMLHTTP");
       } catch (e) {
           try {
               request = new ActiveXObject("Microsoft.XMLHTTP");
           } catch (e) {}
       }
   }

   return request;
}

document.getElementById("deploy").addEventListener("submit", function(e){
   e.preventDefault();
```

```
var fromAddress = document.querySelector("#deploy #fromAddress").value;
var privateKey = document.querySelector("#deploy #privateKey").value;
var matchId = document.querySelector("#deploy #matchId").value;
var betAmount = document.querySelector("#deploy #betAmount").value;

var url = "/getURL?matchId=" + matchId;

var request = getAJAXObject();

request.open("GET", url);

request.onreadystatechange = function() {
    if (request.readyState == 4) {
        if (request.status == 200) {
            if(request.responseText != "An error occured")
            {
        var queryURL = request.responseText;

        var contract = web3.eth.contract(bettingContractABI);
        var data = contract.new.getData(matchId,
          web3.toWei(betAmount, "ether"), queryURL, {
            data: bettingContractByteCode
             });

        var gasRequired = web3.eth.estimateGas({ data: "0x" + data
          });

   web3.eth.getTransactionCount(fromAddress, function(error, nonce){

    var rawTx = {
         gasPrice: web3.toHex(web3.eth.gasPrice),
          gasLimit: web3.toHex(gasRequired),
           from: fromAddress,
            nonce: web3.toHex(nonce),
             data: "0x" + data,
              };

   privateKey = EthJS.Util.toBuffer(privateKey, "hex");

    var tx = new EthJS.Tx(rawTx);
    tx.sign(privateKey);

   web3.eth.sendRawTransaction("0x" +
    tx.serialize().toString("hex"), function(err, hash) {
         if(!err)
             {document.querySelector("#deploy #message").
                innerHTML = "Transaction Hash: " + hash + ".
                  Transaction is mining...";
            var timer = window.setInterval(function(){
            web3.eth.getTransactionReceipt(hash, function(err, result){
            if(result)
             {window.clearInterval(timer);
       document.querySelector("#deploy #message").innerHTML =
         "Transaction Hash: " + hash + " and contract address is: " +
             result.contractAddress;}
               })
                }, 10000)
                 }
             else
           {document.querySelector("#deploy #message").innerHTML = err;
             }
           });
          })

          }
           }
       }
   };

   request.send(null);

}, false)

document.getElementById("bet").addEventListener("submit", function(e){
   e.preventDefault();

   var fromAddress = document.querySelector("#bet #fromAddress").value;
   var privateKey = document.querySelector("#bet #privateKey").value;
   var contractAddress = document.querySelector("#bet
#contractAddress").value;
   var team = document.querySelector("#bet #team").value;

   if(team == "Home")
   {
         team = 1;
   }
   else
   {
         team = 2;
   }

   var contract =
web3.eth.contract(bettingContractABI).at(contractAddress);
   var amount = contract.amount();

   var data = contract.betOnTeam.getData(team);
```

```
    var gasRequired = contract.betOnTeam.estimateGas(team, {
          from: fromAddress,
          value: amount,
          to: contractAddress
    })

    web3.eth.getTransactionCount(fromAddress, function(error, nonce){

          var rawTx = {
            gasPrice: web3.toHex(web3.eth.gasPrice),
            gasLimit: web3.toHex(gasRequired),
            from: fromAddress,
            nonce: web3.toHex(nonce),
            data: data,
            to: contractAddress,
            value: web3.toHex(amount)
        };

        privateKey = EthJS.Util.toBuffer(privateKey, "hex");

        var tx = new EthJS.Tx(rawTx);
          tx.sign(privateKey);

          web3.eth.sendRawTransaction("0x" + tx.serialize().toString("hex"),
 function(err, hash) {
                if(!err)
                {
     document.querySelector("#bet #message").innerHTML = "Transaction
       Hash: " + hash;
         }
       else
        {
        document.querySelector("#bet #message").innerHTML = err;
       }
      })
    })
     }, false)

 document.getElementById("find").addEventListener("submit", function(e){
    e.preventDefault();

    var contractAddress = document.querySelector("#find
      #contractAddress").value;
    var contract =
       web3.eth.contract(bettingContractABI).at(contractAddress);

    var matchId = contract.matchId();
    var amount = contract.amount();
    var homeAddress = contract.homeBet();
    var awayAddress = contract.awayBet();

    document.querySelector("#find #message").innerHTML = "Contract balance
 is: " + web3.fromWei(web3.eth.getBalance(contractAddress), "ether") + ",
 Match ID is: " + matchId + ", bet amount is: " + web3.fromWei(amount,
 "ether") + " ETH, " + homeAddress + " has placed bet on home team and " +
 awayAddress + " has placed bet on away team";
 }, false)
```

This is how the preceding code works:

1. At first, we store the contract byte code and ABI in the `bettingContractByteCode` and `bettingContractABI` variables, respectively.
2. Then, we are create a `Web3` instance, which is connected to our testnet node.
3. Then, we have the `getAJAXObject` function (a cross-browser compatible function), which returns an AJAX object.
4. Then, we attach a `submit` event listener to the first form, which is used to deploy the contract. In the event listener's callback, we make a request to the `getURL` endpoint by passing `matchId` to get the encrypted query string. And then, we generate the data to deploy the contract. Then, we find out the `gasRequired`. We use the function object's estimateGas method to calculate the gas required, but you can use the web3.eth.estimateGas method too. They both differ in terms of arguments; that is, in the preceding case, you don't need to pass the transaction data. Remember that `estimateGas` will return the block gas limit if the function call throws an exception. Then, we calculate the nonce. Here, we just use the `getTransactionCount` method instead of the actual procedure we learned earlier. We do this just for simplification of the code. Then, we create the raw transaction, signing it and broadcasting it. Once the transaction is mined, we display the contract address.

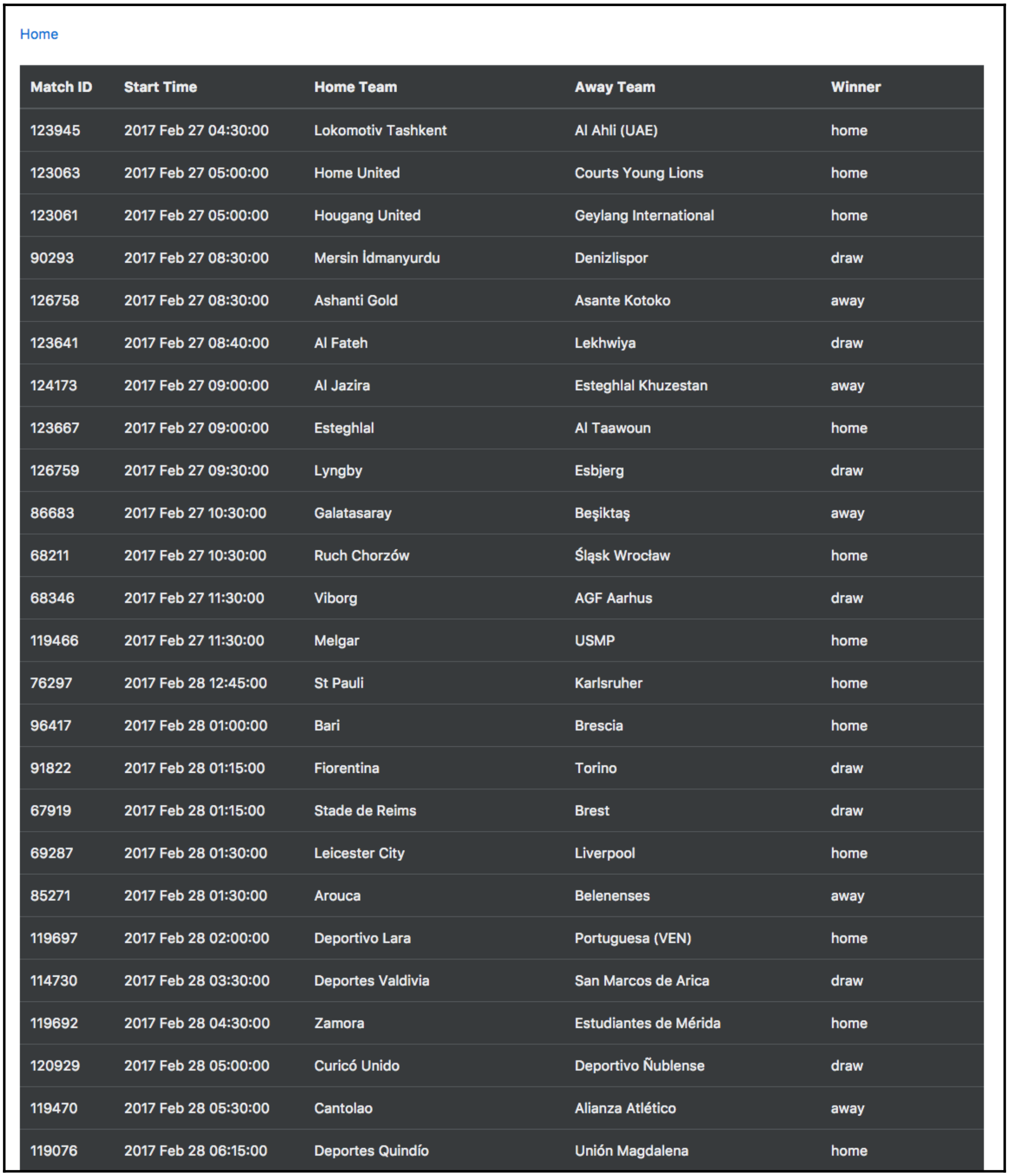
5. Then, we attach a `submit` event listener for the second form, which is used to invest in a contract. Here, we generate the `data` part of the transaction, calculating the gas required, creating the raw transaction, signing it, and broadcasting it. While calculating the gas required for the transaction, we pass the contract address from the account address and value object properties as it's a function call, and the gas differs depending on the value, the from address, and contract address. Remember that while finding the gas required to call a contract's function, you can pass the `to`, `from`,and `value` properties because gas depends on these values.
6. Finally, we have a `submit` event listener for the third form, that is, to display information on a deployed betting contract.

# Testing the client

Now that we have finished building our betting platform, it's time to test it. Before testing, make sure the testnet blockchain is completely downloaded and is looking for new incoming blocks.

Now using our wallet service we built earlier, generate three account. Add one ether to each of the accounts using `http://faucet.ropsten.be:3001/`.

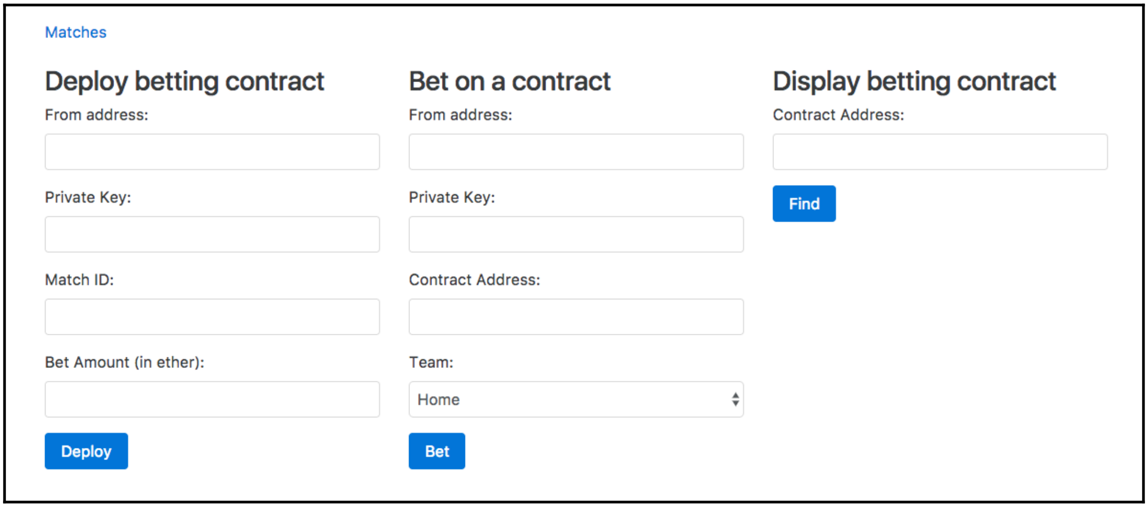
Then, run `node app.js` inside the `Initial` directory and then visit `http://localhost:8080/matches`, and you will see what is shown in this screenshot:



Home

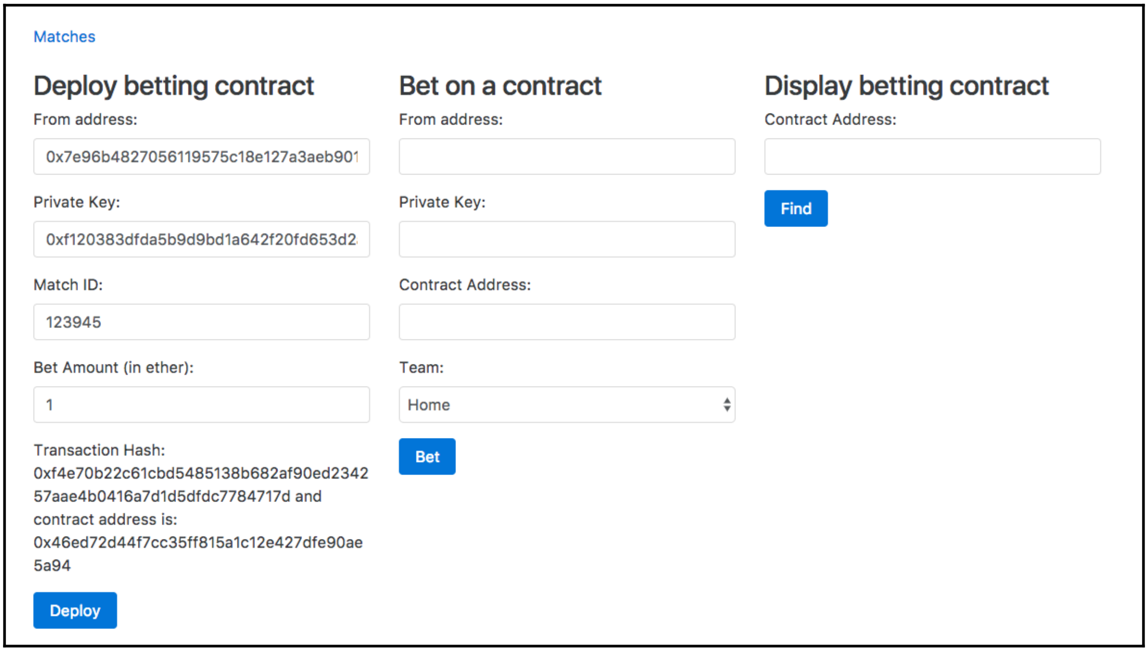
| Match ID | Start Time | Home Team | Away Team | Winner |
|---|---|---|---|---|
| 123945 | 2017 Feb 27 04:30:00 | Lokomotiv Tashkent | Al Ahli (UAE) | home |
| 123063 | 2017 Feb 27 05:00:00 | Home United | Courts Young Lions | home |
| 123061 | 2017 Feb 27 05:00:00 | Hougang United | Geylang International | home |
| 90293 | 2017 Feb 27 08:30:00 | Mersin İdmanyurdu | Denizlispor | draw |
| 126758 | 2017 Feb 27 08:30:00 | Ashanti Gold | Asante Kotoko | away |
| 123641 | 2017 Feb 27 08:40:00 | Al Fateh | Lekhwiya | draw |
| 124173 | 2017 Feb 27 09:00:00 | Al Jazira | Esteghlal Khuzestan | away |
| 123667 | 2017 Feb 27 09:00:00 | Esteghlal | Al Taawoun | home |
| 126759 | 2017 Feb 27 09:30:00 | Lyngby | Esbjerg | draw |
| 86683 | 2017 Feb 27 10:30:00 | Galatasaray | Beşiktaş | away |
| 68211 | 2017 Feb 27 10:30:00 | Ruch Chorzów | Śląsk Wrocław | home |
| 68346 | 2017 Feb 27 11:30:00 | Viborg | AGF Aarhus | draw |
| 119466 | 2017 Feb 27 11:30:00 | Melgar | USMP | home |
| 76297 | 2017 Feb 28 12:45:00 | St Pauli | Karlsruher | home |
| 96417 | 2017 Feb 28 01:00:00 | Bari | Brescia | home |
| 91822 | 2017 Feb 28 01:15:00 | Fiorentina | Torino | draw |
| 67919 | 2017 Feb 28 01:15:00 | Stade de Reims | Brest | draw |
| 69287 | 2017 Feb 28 01:30:00 | Leicester City | Liverpool | home |
| 85271 | 2017 Feb 28 01:30:00 | Arouca | Belenenses | away |
| 119697 | 2017 Feb 28 02:00:00 | Deportivo Lara | Portuguesa (VEN) | home |
| 114730 | 2017 Feb 28 03:30:00 | Deportes Valdivia | San Marcos de Arica | draw |
| 119692 | 2017 Feb 28 04:30:00 | Zamora | Estudiantes de Mérida | home |
| 120929 | 2017 Feb 28 05:00:00 | Curicó Unido | Deportivo Ñublense | draw |
| 119470 | 2017 Feb 28 05:30:00 | Cantolao | Alianza Atlético | away |
| 119076 | 2017 Feb 28 06:15:00 | Deportes Quindío | Unión Magdalena | home |

Here, you can copy any match ID. Let's assume you want to test with the first match, that is, 123945. Now visit `http://localhost:8080` and you will see what is shown in this screenshot:



Team: Home

Now deploy the contract by filling the input fields in the first form and clicking on the **Deploy** button, as shown here. Use your first account to deploy the contract.



Deploy betting contract

From address: 0x7e96b4827056119575c18e127a3aeb901

Private Key: 0xf120383dfda5b9d9bd1a642f20fd653d2

Match ID: 123945

Bet Amount (in ether): 1

Transaction Hash: 0xf4e70b22c61cbd5485138b682af90ed234257aae4b0416a7d1d5dfdc7784717d and contract address is: 0x46ed72d44f7cc35ff815a1c12e427dfe90ae5a94

Bet on a contract

From address:

Private Key:

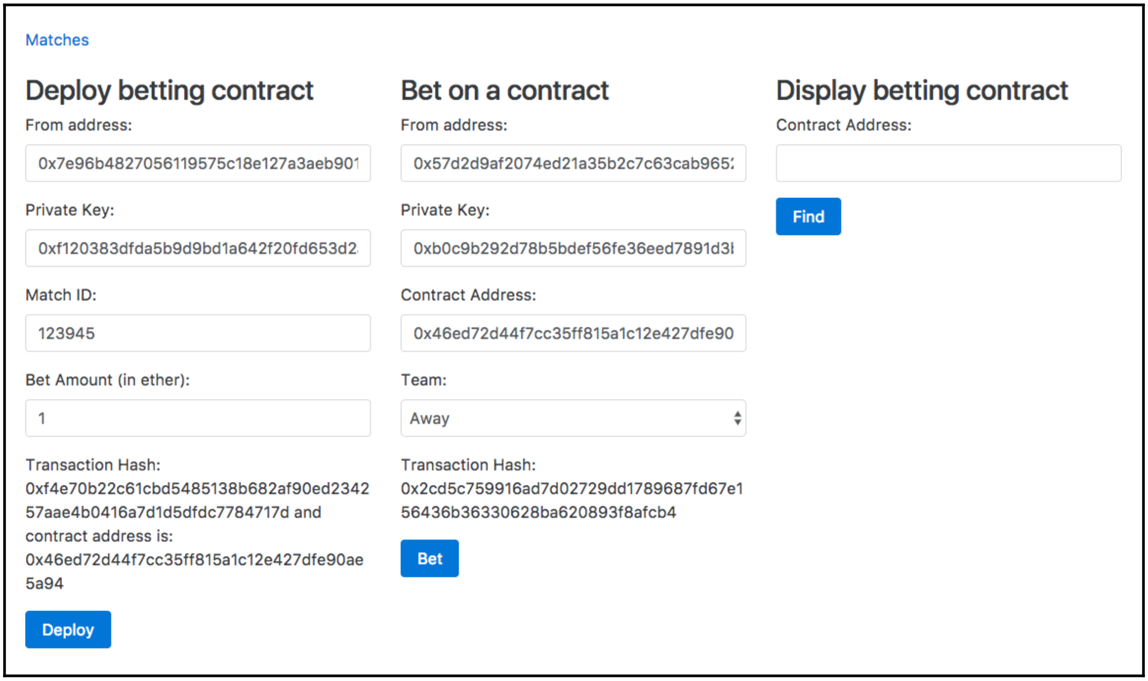
Contract Address:

Team: Home

Display betting contract

Contract Address:

Now bet on the contract's home team from the second account and the `away team` from the third account , as shown in the following screenshot:



Deploy betting contract

From address:

0x7e96b4827056119575c18e127a3aeb901

Private Key:

0xf120383dfda5b9d9bd1a642f20fd653d2

Match ID:

123945

Bet Amount (in ether):

1

Transaction Hash: 0xf4e70b22c61cbd5485138b682af90ed234257aae4b0416a7d1d5dfdc7784717d and contract address is: 0x46ed72d44f7cc35ff815a1c12e427dfe90ae5a94

Bet on a contract

From address:

0x57d2d9af2074ed21a35b2c7c63cab965

Private Key:

0xb0c9b292d78b5bdef56fe36eed7891d3

Contract Address:

0x46ed72d44f7cc35ff815a1c12e427dfe90

Team:

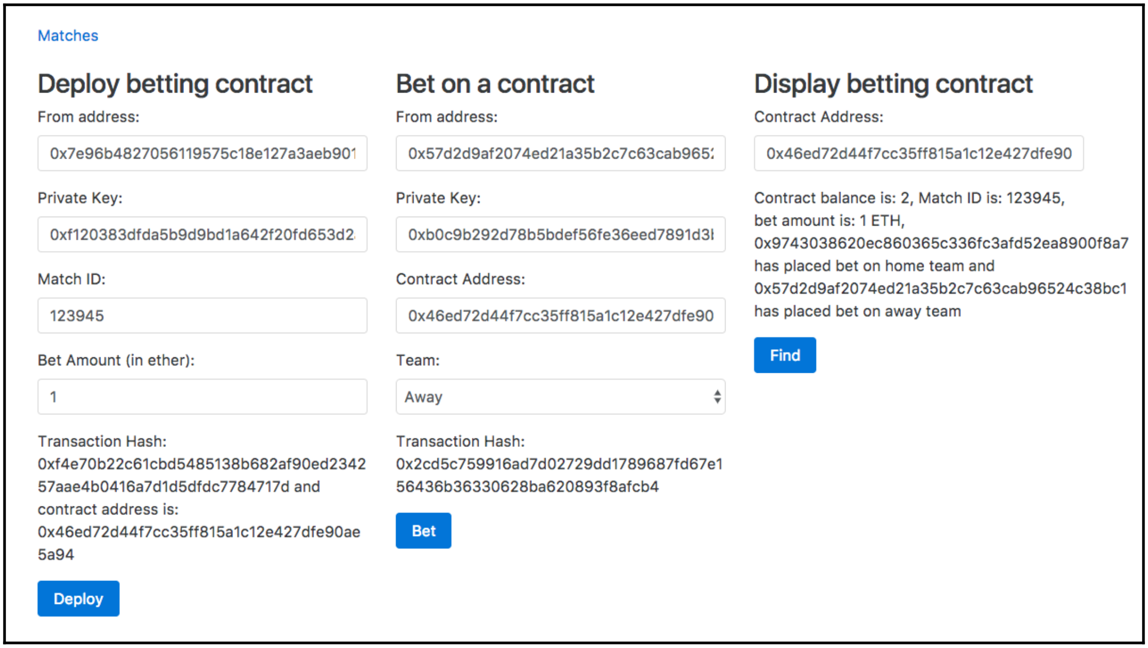
Away

Transaction Hash: 0x2cd5c759916ad7d02729dd1789687fd67e156436b36330628ba620893f8afcb4

Display betting contract

Contract Address:

Now put the contract address on the third form and click on the **Find** button to see the details about the contract. You will see something similar to what is shown in the following screenshot:



Deploy betting contract

From address: 0x7e96b4827056119575c18e127a3aeb901

Private Key: 0xf120383dfda5b9d9bd1a642f20fd653d2

Match ID: 123945

Bet Amount (in ether): 1

Transaction Hash: 0xf4e70b22c61cbd5485138b682af90ed234257aae4b0416a7d1d5dfdc7784717d and contract address is: 0x46ed72d44f7cc35ff815a1c12e427dfe90ae5a94

Bet on a contract

From address: 0x57d2d9af2074ed21a35b2c7c63cab965

Private Key: 0xb0c9b292d78b5bdef56fe36eed7891d3

Contract Address: 0x46ed72d44f7cc35ff815a1c12e427dfe90

Team: Away

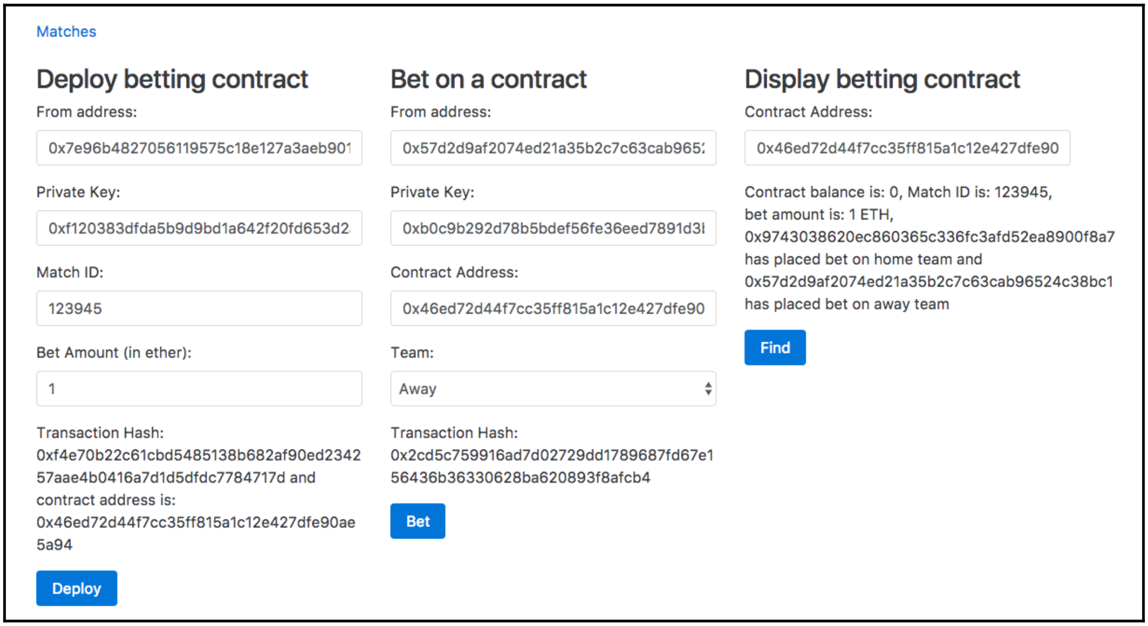
Transaction Hash: 0x2cd5c759916ad7d02729dd1789687fd67e156436b36330628ba620893f8afcb4

Display betting contract

Contract Address: 0x46ed72d44f7cc35ff815a1c12e427dfe90

Contract balance is: 2, Match ID is: 123945, bet amount is: 1 ETH, 0x9743038620ec860365c336fc3afd52ea8900f8a7 has placed bet on home team and 0x57d2d9af2074ed21a35b2c7c63cab96524c38bc1 has placed bet on away team

Once both the transactions are mined, check the details of the contract again, and you will see something similar to what is shown in the following screenshot:



Deploy betting contract

From address: 0x7e96b4827056119575c18e127a3aeb901

Private Key: 0xf120383dfda5b9d9bd1a642f20fd653d2

Match ID: 123945

Bet Amount (in ether): 1

Transaction Hash: 0xf4e70b22c61cbd5485138b682af90ed234257aae4b0416a7d1d5dfdc7784717d and contract address is: 0x46ed72d44f7cc35ff815a1c12e427dfe90ae5a94

Bet on a contract

From address: 0x57d2d9af2074ed21a35b2c7c63cab965

Private Key: 0xb0c9b292d78b5bdef56fe36eed7891d3

Contract Address: 0x46ed72d44f7cc35ff815a1c12e427dfe90

Team: Away

Transaction Hash: 0x2cd5c759916ad7d02729dd1789687fd67e156436b36330628ba620893f8afcb4

Display betting contract

Contract Address: 0x46ed72d44f7cc35ff815a1c12e427dfe90

Contract balance is: 0, Match ID is: 123945, bet amount is: 1 ETH, 0x9743038620ec860365c336fc3afd52ea8900f8a7 has placed bet on home team and 0x57d2d9af2074ed21a35b2c7c63cab96524c38bc1 has placed bet on away team

Here, you can see that the contract doesn't have any ether and all the ether was transferred to the account that put the bet on the home team.

# Summary

In this chapter, we learned about Oraclize and the `strings` library in depth. We used them together to build a decentralized betting platform. Now you can go ahead and customize the contract and the client based on your requirements. To enhance the app, you can add events to the contract and display notifications on the client. The objective was to understand the basic architecture of a decentralized betting app.

In the next chapter, we will learn how to build enterprise-level Ethereum smart contracts using truffle by building our own crypto currency.

# 8
# Building Enterprise Level Smart Contracts

Until now, we were using browser Solidity to write and compile Solidity code. And we were testing our contracts using web3.js. We could have also used the Solidity online IDE to test them. This seemed alright as we were only compiling a single small contract and it had very few imports. As you start building large and complicated smart contracts, you will start facing problems with compiling and testing using the current procedure. In this chapter, we will learn about truffle, which makes it easy to build enterprise-level DApps, by building an altcoin. All the crypto-currencies other than bitcoin are called altcoins.

In this chapter, we'll cover the following topics:

- What the `ethereumjs-testrpc` node is and how to use it?
- What are topics of events?
- Working with contracts using the `truffle-contract` package.
- Installing truffle and exploring the truffle command-line tool and configuration file
- Compiling, deploying, and testing Solidity code using truffle
- Package management via NPM and EthPM
- Using the truffle console and writing external scripts
- Building clients for the DApp using truffle

# Exploring ethereumjs-testrpc

`ethereumjs-testrpc` is a Node.js-based Ethereum node used for testing and development. It simulates full-node behavior and makes the development of Ethereum applications much faster. It also includes all popular RPC functions and features (such as events) and can be run deterministically to make development a breeze.

It's written in JavaScript and is distributed as an `npm` package. At the time of writing this, the latest version of `ethereumjs-testrpc` is 3.0.3 and requires at least Node.js version 6.9.1 to run properly.

It holds everything in memory; therefore, whenever the node is restarted, it loses the previous state.

# Installation and usage

There are three ways to simulate an Ethereum node using `ethereumjs-testrpc`. Each of these ways has its own use cases. Let's explore them.

## The testrpc command-line application

The `testrpc` command can be used to simulate an Ethereum node. To install this command-line app, you need to install `ethereumjs-testrpc` globally:

```
npm install -g ethereumjs-testrpc
```

Here are the various options that can be provided:

- `-a` or `--accounts`: This specifies the number of accounts to be generated at startup.
- `-b` or `--blocktime`: This specifies the blocktime in seconds for automatic mining. The default is 0, and there's no auto-mining.
- `-d` or `--deterministic`: Whenever the node is run, it will generate 10 deterministic addresses; that is, when you provide this flag, the same set of addresses are generated every time. This option can be used to generate deterministic addresses based on a predefined mnemonic as well.

- `-n` or `--secure`: Locks the available accounts by default. When this option is used without the `--unlock` option, the HD wallet will not be created.
- `-m` or `--mnemonic`: Uses a specific HD wallet mnemonic to generate initial addresses.
- `-p` or `--port`: The port number to listen on. Defaults to 8545.
- `-h` or `--hostname`: The hostname to listen on. Defaults to Node's `server.listen()` default.
- `-s` or `--seed`: The arbitrary data to generate the HD wallet mnemonic to be used.
- `-g` or `--gasPrice`: Uses a custom gas price (defaults to 1). If the gas price is not provided while sending the transaction to the node, then this gas price is used.
- `-l` or `--gasLimit`: Uses a custom limit (defaults to 0x47E7C4). If the gas limit is not provided while sending the transaction to node, then this gas limit is used.
- `-f` or `--fork`: This is the fork from another currently running Ethereum node at a given block. The input should be the HTTP location and port of the other client; for example, `http://localhost:8545`. Optionally, you can specify the block to fork from using an @ sign: `http://localhost:8545@1599200`.
- `--debug`: Outputs VM opcodes for debugging.
- `--account`: This option is used to import accounts. It specifies `--account=...` any number of times, passing arbitrary private keys and their associated balances to generate initial addresses. An `testrpc --account="privatekey,balance" [--account="privatekey,balance"]` an HD wallet will not be created for you when using `--account`.
- `-u` or `--unlock`: Specifies `--unlock ...` any number of times, passing either an address or an account index to unlock specific accounts. When used in conjunction with `--secure`, `--unlock` will override the locked state of the specified accounts: `testrpc --secure --unlock "0x1234..." --unlock "0xabcd..."`. You can also specify a number, unlocking accounts by their index: `testrpc --secure -u 0 -u 1`. This feature can also be used to impersonate accounts and unlock addresses you wouldn't otherwise have access to. When used with the `--fork` feature, you can use the `testrpc` to make transactions as any address on the blockchain, which is very useful in testing and dynamic analysis.
- `--networkId`: Used to specify a network ID that this node is part of.

Note that private keys are 64 characters long and must be input as a 0x-prefixed hex string. The balance can either be input as an integer or a 0x-prefixed hex value specifying the amount of wei in that account.

## Using ethereumjs-testrpc as a web3 provider or as an HTTP server

You can use `ethereumjs-testrpc` as a `web3` provider like this:

```
var TestRPC = require("ethereumjs-testrpc");
web3.setProvider(TestRPC.provider());
```

You can use `ethereumjs-testrpc` as a general HTTP server like this:

```
var TestRPC = require("ethereumjs-testrpc");
var server = TestRPC.server();
server.listen(port, function(err, blockchain) {});
```

Both `provider()` and `server()` take a single object that allows you to specify the behavior of the `ethereumjs-testrpc`. This parameter is optional. The available options are as follows:

- `accounts`: Value is an array of objects. Each object should have a balance key with a hexadecimal value. The secretKey key can also be specified, which represents the account's private key. If there's no secretKey, the address is autogenerated with the given balance. If specified, the key is used to determine the account's address.
- `debug`: Outputs VM opcodes for debugging.
- `logger`: Value is an object that implements a `log()` function.
- `mnemonic`: Uses a specific HD wallet mnemonic to generate initial addresses.
- `port`: The port number to listen on when running as a server.
- `seed`: Arbitrary data to generate the HD wallet mnemonic to be used.
- `total_accounts`: The number of accounts to generate at start up.
- `fork`: The same as the preceding `--fork` option.
- `network_id`: The same as the `--networkId` option. Used to specify a network ID that this node is part of.
- `time`: The date that the first block should start. Use this feature along with the `evm_increaseTime` method to test `time-dependent` code.
- `locked`: Specifies whether or not accounts are locked by default.
- `unlocked_accounts`: An array of addresses or address indexes specifying which accounts should be unlocked.

## Available RPC methods

Here is the list of RPC methods made available with `ethereumjs-testrpc`:

- `eth_accounts`
- `eth_blockNumber`
- `eth_call`
- `eth_coinbase`
- `eth_compileSolidity`
- `eth_estimateGas`
- `eth_gasPrice`
- `eth_getBalance`
- `eth_getBlockByNumber`
- `eth_getBlockByHash`
- `eth_getBlockTransactionCountByHash`
- `eth_getBlockTransactionCountByNumber`
- `eth_getCode (only supports block number "latest")`
- `eth_getCompilers`
- `eth_getFilterChanges`
- `eth_getFilterLogs`
- `eth_getLogs`
- `eth_getStorageAt`
- `eth_getTransactionByHash`
- `eth_getTransactionByBlockHashAndIndex`
- `eth_getTransactionByBlockNumberAndIndex`
- `eth_getTransactionCount`
- `eth_getTransactionReceipt`
- `eth_hashrate`
- `eth_mining`
- `eth_newBlockFilter`
- `eth_newFilter (includes log/event filters)`
- `eth_sendTransaction`
- `eth_sendRawTransaction`
- `eth_sign`
- `eth_syncing`

- `eth_uninstallFilter`
- `net_listening`
- `net_peerCount`
- `net_version`
- `miner_start`
- `miner_stop`
- `rpc_modules`
- `web3_clientVersion`
- `web3_sha3`

There are also special nonstandard methods that aren't included within the original RPC specification:

- `evm_snapshot`: Snapshots the state of the blockchain at the current block. Takes no parameters. Returns the integer ID of the snapshot created.
- `evm_revert`: Reverts the state of the blockchain to a previous snapshot. Takes a single parameter, which is the snapshot ID to revert to. If no snapshot ID is passed, it will revert to the latest snapshot. Returns true.
- `evm_increaseTime`: Jumps forward in time. Takes one parameter, which is the amount of time to increase in seconds. Returns the total time adjustment in seconds.
- `evm_mine`: Forces a block to be mined. Takes no parameters. Mines a block independent of whether or not mining is started or stopped.

# What are event topics?

Topics are values used for indexing events. You cannot search for events without topics. Whenever an event is invoked, a default topic is generated, which is considered the first topic of the event. There can be up to four topics for an event. Topics are always generated in the same order. You can search for an event using one or more of its topics.

The first topic is the signature of the event. The rest of the three topics are the values of indexed parameters. If the index parameter is `string`, `bytes`, or `array`, then the keccak-256 hash of it is the topic instead.

Let's take an example to understand topics. Suppose there is an event of this form:

```
event ping(string indexed a, int indexed b, uint256 indexed c, string d,
int e);

//invocation of event
ping("Random String", 12, 23, "Random String", 45);
```

Here, these four topics are generated. They are as follows:

- `0xb62a11697c0f56e93f3957c088d492b505b9edd7fb6e7872a93b41cdb2020644`: This is the first topic. It is generated using `web3.sha3("ping(string,int256,uint256,string,int256)")`. Here, you can see that all types are of a canonical form.
- `0x30ee7c926ebaf578d95b278d78bc0cde445887b0638870a26dcab901ba21d3f2`: This is the second topic. It is generated using `web3.sha3("Random String")`.
- The third and fourth topics are `0x000000000000000000000000000000000000000000000000000000000000000c` and `0x0000000000000000000000000000000000000000000000000000000000000017`, respectively, that is, hexadecimal representation of the values . They are calculated using `EthJS.Util.bufferToHex(EthJS.Util.setLengthLeft(12, 32))` and `EthJS.Util.bufferToHex(EthJS.Util.setLengthLeft(23, 32))`, respectively.

Internally, your Ethereum node will build indexes using topics so that you can easily find events based on signatures and indexed values.

Suppose you want to get event calls of the preceding event, where the first argument is `Random String` and the third argument is either `23` or `78`; then, you can find them using `web3.eth.getFilter` this way:

```
var filter = web3.eth.filter({
   fromBlock: 0,
   toBlock: "latest",
   address: "0x853cdcb4af7a6995808308b08bb78a74de1ef899",
   topics:
["0xb62a11697c0f56e93f3957c088d492b505b9edd7fb6e7872a93b41cdb2020644",
"0x30ee7c926ebaf578d95b278d78bc0cde445887b0638870a26dcab901ba21d3f2", null,
[EthJS.Util.bufferToHex(EthJS.Util.setLengthLeft(23, 32)),
EthJS.Util.bufferToHex(EthJS.Util.setLengthLeft(78, 32))]]
});
```

```
filter.get(function(error, result){
 if (!error)
   console.log(result);
});
```

So here, we are asking the node to return all events from the blockchain that have been fired by the `0x853cdcb4af7a6995808308b08bb78a74de1ef899` contract address, whose first topic is `0xb62a11697c0f56e93f3957c088d492b505b9edd7fb6e7872a93b41cdb2020644`, the second topic is `0x30ee7c926ebaf578d95b278d78bc0cde445887b0638870a26dcab901ba21d3f2`, and the third topic is either `0x0000000000000000000000000000000000000000000000000000000000000017` or `0x000000000000000000000000000000000000000000000000000000000000004e`.

In the preceding code, note the order of the `topics` array values. The order is important.

# Getting started with truffle-contract

It is important to learn `truffle-contract` before learning truffle because `truffle-contract` is tightly integrated into truffle. Truffle tests, code to interact with contracts in truffle, deployment code, and so on are written using `truffle-contract`.

The `truffle-contract` API is a JavaScript and Node.js library, which makes it easy to work with ethereum smart contracts. Until now, we have been using web3.js to deploy and call smart contracts functions, which is fine, but `truffle-contract` aims to make it even easier to work with ethereum smart contracts. Here are some features of `truffle-contract` that make it a better choice then web3.js in order to work with smart contracts:

- Synchronized transactions for better control flow (that is, transactions won't finish until you're guaranteed they've been mined).
- Promise-based API. No more callback hell. Works well with ES6 and async/await.
- Default values for transactions, such as from `address` or `gas`.
- Returning logs, transaction receipt, and transaction hash of every synchronized transaction.

Before we get into `truffle-contract`, you need to know that it doesn't allow us to sign transactions using accounts stored outside of the ethereum node; that is, it doesn't have anything similar to `sendRawTransaction`. The `truffle-contract` API assumes that every user of your DApp has their own ethereum node running and they have their accounts stored in that node. Actually this is how DApps should work because if every DApp's client starts letting users create and manage accounts, then it will be a concern for users to manage so many accounts and painful for developers to develop a wallet manager every time for every client they build. Now, the question is how will clients know where the user has stored the accounts and in what format? So, for portability reasons, it's recommended that you assume that users have their accounts stored in their personal node, and to manage the account, they use something like the ethereum Wallet app. As accounts stored in the Ethereum node are signed by the ethereum node itself, there is no need for `sendRawTransaction` anymore. Every user needs to have their own node and cannot share a node because when an account is unlocked, it will be open for anyone to use it, which will enable users to steal other's ether and make transactions from others' accounts.

If you are using an app that requires you to host your own node and manage accounts in it, then make sure you don't allow everyone to make JSON-RPC calls to that node; instead, only local apps should be able to make calls. Also, make sure that you don't keep the accounts unlocked for very long and lock them as soon as you don't need the account.

If your applications require the functionality of creating and signing raw transactions, then you can use `truffle-contract` just to develop and test smart contracts, and in your application, you can interact with contracts just like we were doing earlier.

# Installing and importing truffle-contract

At the time of writing this, the latest version of the `truffle-contract` API is 1.1.10. Before importing `truffle-contract`, you need to first import web3.js as you will need to create a provider to work with the `truffle-contract` APIs so that `truffle-contract` will internally use the provider to make JSON-RPC calls.

To install `truffle-contract` in the Node.js app, you need to simply run this in your app directory:

```
npm install truffle-contract
```

And then use this code to import it:

```
var TruffleContract = require("truffle-contract");
```

To use truffle-contract in a browser, you can find the browser distribution inside the dist directory in the https://github.com/trufflesuite/truffle-contract repository.

In HTML, you can enqueue it this way:

```
<script type="text/javascript" src="./dist/truffle-
contract.min.js"></script>
```

Now you will have a TruffleContract global variable available.

# Setting up a testing environment

Before we start learning about truffle-contract APIs, we need to set up a testing environment, which will help us test our code while learning.

First of all, run the ethereumjs-testrpc node representing network ID 10 by just running the testrpc --networkId 10 command. We have randomly chosen network ID 10 for development purposes, but you are free to choose any other network ID. Just make sure it's not 1 as mainnet is always used in live apps and not for development and testing purposes.

Then, create an HTML file and place this code in it:

```
<!doctype html>
<html>
   <body>
         <script type="text/javascript" src="./web3.min.js"></script>
         <script type="text/javascript" src="./truffle-
            contract.min.js"></script>
         <script type="text/javascript">
               //place your code here
         </script>
   </body>
</html>
```

Download web3.min.js and truffle-contract.min.js. You can find the truffle-contract browser build at https://github.com/trufflesuite/truffle-contract/tree/master/dist.

# The truffle-contract API

Now let's explore `truffle-contract` APIs. Basically, `truffle-contract` has two APIs, that is, the contract abstraction API and the contract instance API. A contract abstraction API represents various kinds of information about the contract (or a library), such: as its ABI; unlinked byte code; if the contract is already deployed, then its address in various ethereum networks; addresses of the libraries it depends on for various ethereum networks if deployed; and events of the contract. The abstraction API is a set of functions that exist for all contract abstractions. A contract instance represents a deployed contract in a specific network. The instance API is the API available to contract instances. It is created dynamically based on functions available in your `Solidity` source file. A contract instance for a specific contract is created from a contract abstraction that represents the same contract.

## The contract abstraction API

The contract abstraction API is something that makes `truffle-contract` very special compared to web3.js. Here is why it's special:

- It will automatically fetch default values, such as library addresses, contract addresses, and so on, depending on which network it's connected to; therefore, you don't have to edit the source code every time you change the network.
- You may choose to listen to certain events in certain networks only.
- It makes it easy to link libraries to contract's byte code at runtime. There are several other benefits you will find out once you have explored how to use the API.

Before we get into how to create a contract abstraction and its methods, let's write a sample contract, which the contract abstraction will represent. Here is the sample contract:

```
pragma Solidity ^0.4.0;

import "github.com/pipermerriam/ethereum-string-
utils/contracts/StringLib.sol";

contract Sample
{
    using StringLib for *;

    event ping(string status);

    function Sample()
    {
        uint a = 23;
        bytes32 b = a.uintToBytes();

        bytes32 c = "12";
        uint d = c.bytesToUInt();

        ping("Conversion Done");
    }
}
```

This contract converts `uint` into `bytes32` and `bytes32` into `uint` using the `StringLib` library. `StringLib` is available at the `0xcca8353a18e7ab7b3d094ee1f9ddc91bdf2ca6a4` address on the main network, but on other networks, we need to deploy it to test the contract. Before you proceed further, compile it using browser Solidity, as you will need the ABI and byte code.

Now let's create a contract abstraction representing the `Sample` contract and the `StringLib` library. Here is the code for this. Place it in the `HTML` file:

```
var provider = new Web3.providers.HttpProvider("http://localhost:8545");
var web3 = new Web3(provider);

var SampleContract = TruffleContract({
   abi:
[{"inputs":[],"payable":false,"type":"constructor"},{"anonymous":false,"inp
uts":[{"indexed":false,"name":"status","type":"string"}],"name":"ping","typ
e":"event"}],
   unlinked_binary:
"6060604052341561000c57fe5b5b6000600060006000601793508373__StringLib__6394e
8767d90916000604051602001526040518263ffffffff167c01000000000000000000000000
000000000000000000000000000000000281526004018082815260200191505060206040518
083038186803b151561008b57fe5b60325a03f4151561009857fe5b50505060405180519050
92507f313200000000000000000000000000000000000000000000000000000000000091508
16000191673__StringLib__6381a33a6f90916000604051602001526040518263ffffffff1
67c010000000000000000000000000000000000000000000000000000000002815260040180
826000191660001916815260200191505060206040518083038186803b151561014557fe5b6
0325a03f4151561015257fe5b5050506040518051905090507f3adb191b3dee3c3ccbe8c657
275f608902f13e3a020028b12c0d825510439e5660405180806020018281038252600f81526
02001807f436f6e76657273696f6e20446f6e650000000000000000000000000000000000081
525060200191505060405180910390a15b505050505b6033806101da6000396000f30060606
040525bfe00a165627a7a7230582056ebda5c1e4ba935e5ad61a271ce8d59c95e0e4bca4ad2
0e7f07d804801e95c60029",
   networks: {
         1: {
         links: {
       "StringLib": "0xcca8353a18e7ab7b3d094ee1f9ddc91bdf2ca6a4"
               },
        events: {
"0x3adb191b3dee3c3ccbe8c657275f608902f13e3a020028b12c0d825510439e56": {
                "anonymous": false,
                "inputs": [
                         {
                          "indexed": false,
                          "name": "status",
                          "type": "string"
                         }
                           ],
                           "name": "ping",
                           "type": "event"
                     }
               }
         },
         10: {
               events: {
"0x3adb191b3dee3c3ccbe8c657275f608902f13e3a020028b12c0d825510439e56": {
                           "anonymous": false,
                           "inputs": [
                                 {
                                       "indexed": false,
                                       "name": "status",
                                       "type": "string"
                                 }
                           ],
                           "name": "ping",
                           "type": "event"
                     }
               }
         }
   },
   contract_name: "SampleContract",
});

SampleContract.setProvider(provider);
SampleContract.detectNetwork();

SampleContract.defaults({
   from: web3.eth.accounts[0],
   gas: "900000",
   gasPrice: web3.eth.gasPrice,
})

var StringLib = TruffleContract({
   abi:
[{"constant":true,"inputs":[{"name":"v","type":"bytes32"}],"name":"bytesToU
Int","outputs":[{"name":"ret","type":"uint256"}],"payable":false,"type":"fu
```

```
nction"},{"constant":true,"inputs":[{"name":"v","type":"uint256"}],"name":"
uintToBytes","outputs":[{"name":"ret","type":"bytes32"}],"payable":false,"t
ype":"function"}],
   unlinked_binary:
"60606040523415610 00c57fe5b5b6102178061001c6000396000f30060606040526000357c
0100000000000000000000000000000000000000000000000000000000900463ffffffff168
06381a33a6f14610046578063 94e8767d14610076575bfe5b61006060048080356000191690
602001909190505061 00aa565b60405180828152602001915050604051809103 90f35b61008
c6004808035906020019091905050610140565b604051808260001916600019168152602001
91505060405180910390f35b6000600060006000600102846000191614156100c5576100005
65b600090505b6020811015610135576 0ff81601f0360080260020a85600190048115156100
ed57fe5b041691506000821415610 0ff57610135565b6030821080610 10e5750603982115b1
56101185761000 0565b5b600a830292506030820383019250 5b80806001019150506100ca56
5b8292505b5050919050565b6000600082141561017357 7f300000000000000000000000000
0000000000000000000000000000000000090506101e2565b5b600082111561 01e1576101
008160019004811515610 18e57fe5b0460010290507f01000000000000000000000000000000
0000000000000000000000000000000006030600a848115156101c357fe5b06010260010281
179050600a828115156101d957fe5b04915061 0174565b5b8090505b9190505600a165627a7
a72305820d2897c98df4e1a3a71aefc5c486aed29c47c80cfe77e38328ef5f4cb5efcf2f100
29",
   networks: {
         1: {
               address: "0xcca8353a18e7ab7b3d094ee1f9ddc91bdf2ca6a4"
         }
   },
   contract_name: "StringLib",
})


StringLib.setProvider(provider);
StringLib.detectNetwork();

StringLib.defaults({
   from: web3.eth.accounts[0],
   gas: "900000",
   gasPrice: web3.eth.gasPrice,
})
```

Here is how the preceding code works:

1. At first, we create a provider. Using this provider, `truffle-contract` will communicate with the node.

2. Then, we create a contract abstraction for the `Sample` contract. To create a contract abstraction, we use the `TruffleContract` function. This function takes an object, which contains various kinds of information about the contract. This object can be termed as an artifacts object. The `abi` and `unlinked_binary` properties are compulsory. The other properties of the object are optional. The `abi` property points to the ABI of the contract, whereas the `unlinked_binary` property points to the unlinked binary code of the contract.
3. Then, we have a property network that indicates various kinds of information about the contract in various networks. Here, we are saying that in network ID 1, the `StringLib` dependency is deployed at the `0xcca8353a18e7ab7b3d094ee1f9ddc91bdf2ca6a4` address so that at the time of deploying the `Sample` contract in network 1, it will link it automatically. Under a network object, we can also put an `address` property, indicating that the contract is already deployed to this network and this is the contract address. We also have an `events` objects in the `networks` object, which specifies the events of the contract we are interested in catching. The keys of the `events` object are topics of events and the `values` are the ABI of events.
4. Then, we call the `setProvider` method of the `SampleContract` object by passing a new provider instance. This is a way to pass the provider so that `truffle-contract` can communicate with the node. The `truffle-contract` API doesn't provide a way to set the provider globally; instead, you need to set a provider for every contract abstraction. This is a feature that lets us connect and work on multiple networks at once with ease.
5. Then, we call the `detectNetwork` method of the `SampleContract` object. This is the way to set the network ID that the contract abstraction is currently representing; that is, during all the operations on the contract abstraction, the values mapped to this network ID are used. This method will automatically detect which network ID our node is connected to and will set it automatically. If you want to manually set the network ID or change it at runtime, then you can use `SampleContract.setNetwork(network_id)`. If you change the network ID, then make sure that the provider is also pointing to the node of the same network since `truffle-contract` won't be able to map the network ID with correct links, addresses, and events otherwise.
6. Then, we set default values for transactions made for `SampleContract`. This method gets and, optionally, sets transaction defaults. If called without any parameters, it will simply return an object representing the current defaults. If an object is passed, this will set new defaults.
7. We did the same for the `StringLib` library in order to create a contract abstraction for it.

## Creating contract instances

A contract instance represents a deployed contract in a particular network. Using a contract abstraction instance, we need to create a contract instance. There are three methods to create a contract instance:

- `SampleContract.new([arg1, arg2, ...], [tx params])`: This function takes whatever constructor parameters your contract requires and deploys a new instance of the contract to the network to which the contract abstraction is set to use. There's an optional last argument, which you can use to pass transaction parameters, including the transaction from address, gas limit, and gas price. This function returns a promise that resolves into a new instance of the contract abstraction at the newly deployed address when the transaction is mined. This method doesn't make any changes to the artifacts object the contract abstraction represents. Before using this method, make sure that it can find the libraries' addresses that the byte code is dependent on for the network it's set to use.
- `SampleContract.at(address)`: This function creates a new instance of the contract abstraction representing the contract at the passed-in address. It returns a "thenable" object (not yet an actual promise for backward compatibility). It resolves to a contract abstraction instance after ensuring that the code exists at the specified address in the network it's set to use.
- `SampleContract.deployed()`: This is just like `at()`, but the address is retrieved from the artifacts object. `Like at()`, `deployed()` is tenable and will resolve to a contract instance representing the deployed contract after ensuring that the code exists at that location and that the address exists on the network that the contract abstraction is set to use.

Let's deploy and get a contract instance of the `Sample` contract. In `network ID 10`, we need to use `new()` to deploy the `StringLib` library first and then add the deployed address of the `StringLib` library to the `StringLib` abstraction, link the `StringLib` abstraction to the `SampleContract` abstraction, and then deploy the `Sample` contract using `new()` to get an instance of the `Sample` contract. But in `network ID 1`, we just need to deploy `SampleContract` and get its instance, as we already have `StringLib` deployed there. Here is the code to do all this:

```
web3.version.getNetwork(function(err, network_id) {
   if(network_id == 1)
   {
         var SampleContract_Instance = null;

         SampleContract.new().then(function(instance){
               SampleContract.networks[SampleContract.network_id]
                  ["address"] = instance.address;
```

```
                SampleContract_Instance = instance;
            })
        }
        else if(network_id == 10)
        {
            var StringLib_Instance = null;
            var SampleContract_Instance = null;

            StringLib.new().then(function(instance){
                StringLib_Instance = instance;
            }).then(function(){
                StringLib.networks[StringLib.network_id] = {};
                StringLib.networks[StringLib.network_id]["address"] =
                  StringLib_Instance.address;
                SampleContract.link(StringLib);
            }).then(function(result){
                return SampleContract.new();
            }).then(function(instance){
                SampleContract.networks[SampleContract.network_id]
                  ["address"] = instance.address;
                SampleContract_Instance = instance;
            })
        }
    });
```

This is how the preceding code works:

1. At first, we detect the network ID. If the network ID is `10`, then we deploy both the contract and library, and if the network ID is `10`, then we only deploy the contract.
2. In `network ID 10`, we deploy the `StringLib` contract and get the contract instance of it.
3. Then, we update the `StringLib` abstraction so that it knows about the address of the contract in the current network it represents. The interface to update the abstraction is similar to updating the artifacts object directly. If you are connected to network ID 1, then it will override the `StringLib` address, which is already set.
4. Then, we link the deployed `StringLib` to the `SampleContract` abstraction. Linking updates the links and copies the events of the library to the `SampleContract` abstraction's current network it represents. Libraries can be linked multiple times and will overwrite their previous linkage.

5. We deploy `SampleContract` to the current network.
6. We update the `SampleContract` abstraction to store the address of the contract in the current network it's representing so that we can use `deployed()` to get the instance later on.
7. In the case of network ID 1, we just deploy `SampleContract` and that's it.
8. Now you can simply change the network that your node is connected to and restart your app, and your app will behave accordingly. So for example, on a developer's machine, the app will be connected to a development network and on a production server, it will be connected to the main network. Obviously, you may not want to deploy the contracts every time the preceding file is run, so you can actually update the artifacts objects once the contracts are deployed and in the code you can check whether the contract is deployed or not. If not deployed, only then should you deploy it. Instead of updating the artifacts object manually, you can store the artifacts in a DB or in a file and write code to update them automatically after the contract deployment is done.

## The contract instance API

Each contract instance is different based on the source Solidity contract, and the API is created dynamically. Here are the various the APIs of a contract instance:

- `allEvents`: This is a function of a contract instance that takes a callback that is invoked whenever an event is fired by the contract matching the event signature under the current network ID in the contract artifacts object. You can also use `event name-specific` functions to catch specific events instead of all of them. In the preceding contract, to catch ping events, you can use `SampleContract_Instance.ping(function(e, r){})`.
- `send`: This function is used to send ether to the contract. It takes two arguments; that is, the first argument is the amount of wei to transfer and the second argument is an optional object that can used to set the `from` of the transaction, which indicates from which address the ether is being sent. This call returns a promise, and the promise resolves to the details about the transaction when its mined.

- We can invoke any method of the contract using `SampleContract.functionName()` or `SampleContract.functionName.call()`. The first one sends a transaction, whereas the second one invokes the method on the EVM only, and the changes are not persistent. Both of these methods return a promise. In the first case, the promise resolves to the result of the transaction, that is, an object holding a transaction hash, logs, and transaction receipt. And in the second case, it resolves to the return value of the method `call`. Both the methods take function arguments and an optional last argument, which is an object to set `from`, `gas`, `value`, and so on of the transaction.

# Introduction to truffle

Truffle is a development environment (providing a command-line tool to compile, deploy, test, and build), framework (providing various packages to make it easy to write tests, deployment code, build clients, and so on) and asset pipeline (publishing packages and using packages published by others) to build ethereum-based DApps.

## Installing truffle

Truffle works on OS X, Linux, and Windows. Truffle requires you to have Node.js version 5.0+ installed. At the time of writing this, the latest stable version of truffle is 3.1.2, and we will be using this version. To install truffle, you just need to run this command:

```
npm install -g truffle
```

Before we go ahead, make sure you are running testrpc with network ID 10. The reason is the same as the one discussed earlier.

## Initializing truffle

First, you need to create a directory for your app. Name the directory altcoin. Inside the altcoin directory, run this command to initialize your project:

```
truffle init
```

Once completed, you'll have a project structure with the following items:

- contracts: The directory where truffle expects to find `Solidity` contracts.
- migrations: The directory to place files that contain contract deployment code.
- `test`: The location of test files to test your smart contracts.
- `truffle.js`: The main truffle configuration file.

By default, `truffle init` gives you a set of example contracts (`MetaCoin` and `ConvertLib`), which act like a simple altcoin built on top of ethereum.

Here is the source code of the MetaCoin smart contract just for reference:

```
pragma Solidity ^0.4.4;

import "./ConvertLib.sol";

contract MetaCoin {
   mapping (address => uint) balances;

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

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

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

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

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

MetaCoin assigns 10 k metacoins to the account address that deployed the contract. 10 k is the total amount of bitcoins that exists. Now this user can send these metacoins to anyone using the sendCoin() function. You can find the balance of your account using getBalance()anytime. Assuming that one metacoin is equal to two ethers, you can get the balance in ether using getBalanceInEth().

The  ConvertLib library is used to calculate the value of metacoins in ether. For this purpose, it provides the convert() method.

# Compiling contracts

Compiling contracts in truffle results in generating artifact objects with the abi and unlinked_binary set. To compile, run this command:

```
truffle compile
```

Truffle will compile only the contracts that have been changed since the last compilation in order to avoid any unnecessarily compilation. If you'd like to override this behavior, run the preceding command with the --all  option.

You can find the artifacts in the build/contracts directory. You are free to edit these files according to your needs. These files get modified at the time of running the compile and migrate commands.

Here are a few things you need to take care of before compiling:

- Truffle expects your contract files to define contracts that match their filenames exactly. For instance, if you have a file called MyContract.sol, one of these should exist within the contract file: contract MyContract{} or library myContract{}.
- Filename matching is case-sensitive, which means that if your filename isn't capitalized, your contract name shouldn't be capitalized either.
- You can declare contract dependencies using Solidity's import command. Truffle will compile contracts in the correct order and link libraries automatically when necessary. Dependencies must be specified as relative to the current Solidity file, beginning with either ./ or ../.

Truffle version 3.1.2 uses compiler version 0.4.8. Truffle doesn't currently support changing the compiler version, so it's fixed.

# Configuration files

The truffle.js file is a JavaScript file used to configure the project. This file can execute any code necessary to create the configuration for the project. It must export an object representing your project configuration. Here is the default content of the file:

```
module.exports = {
  networks: {
    development: {
      host: "localhost",
      port: 8545,
      network_id: "*" // Match any network id
    }
  }
};
```

There are various properties this object can contain. But the most basic one is networks. The networks property specifies which networks are available for deployment as well as specific transaction parameters when interacting with each network (such as gasPrice, from, gas, and so on). The default gasPrice is 100,000,000,000, gas is 4712388, and from is the first available contract in the ethereum client.

You can specify as many networks as you want. Go ahead and edit the configuration file to this:

```
module.exports = {
  networks: {
    development: {
      host: "localhost",
      port: 8545,
      network_id: "10"
    },
    live: {
         host: "localhost",
      port: 8545,
      network_id: "1"
    }
  }
};
```

In the preceding code, we are defining two networks with the names development and live.

When using Command Prompt on Windows, the default configuration filename can cause a conflict with the truffle executable. If this is the case, we recommend that you use Windows PowerShell or Git BASH as these shells do not have this conflict. Alternatively, you can rename the configuration file to truffle-config.js in order to avoid this conflict.

# Deploying contracts

Even the smallest project will interact with at least two blockchains: one on the developer's machine, such as the EthereumJS TestRPC, and the other representing the network where the developer will eventually deploy their application (this could be the main ethereum network or a private consortium network, for instance).

Because the network is auto-detected by the contract abstractions at runtime, it means that you only need to deploy your application or frontend once. When your application is run, the running ethereum client will determine which artifacts are used, and this will make your application very flexible.

JavaScript files that contain code to deploy contracts to the ethereum network are called migrations. These files are responsible for staging your deployment tasks, and they're written under the assumption that your deployment needs will change over time. As your project evolves, you'll create new migration scripts to further this evolution on the blockchain. A history of previously run migrations is recorded on the blockchain through a special Migrations contract. If you have seen the contents of the contracts and build/contracts directory, then you would have noticed the Migrations contract's existence there. This contract should always be there and shouldn't be touched unless you know what you are doing.

## Migration files

In the migrations directory, you will notice that the filenames are prefixed with a number; that is, you will find 1_initial_migration.js and 2_deploy_contracts.js files. The numbered prefix is required in order to record whether the migration ran successfully.

The Migrations contract stores (in last_completed_migration) a number that corresponds to the last applied migration script found in the migrations folder. The Migrations contract is always deployed first. The numbering convention is x_script_name.js, with x starting at 1. Your app contracts would typically come in scripts starting at 2.

So, as this `Migrations` contract stores the number of the last deployment script applied, truffle will not run these scripts again. On the other hand, in future, your app may need to have a modified, or new, contract deployed. For that to happen, you create a new script with an increased number that describes the steps that need to take place. Then, again, after they have run once, they will not run again.

## Writing migrations

At the beginning of a migration file, we tell truffle which contracts we'd like to interact with via the `artifacts.require()` method. This method is similar to Node's `require`, but in our case, it specifically returns a contract abstraction that we can use within the rest of our deployment script.

All migrations must export a function via the `module.exports` syntax. The function exported by each migration should accept a `deployer` object as its first parameter. This object assists in deployment both by providing a clear API to deploy smart contracts as well as performing some of the deployment's more mundane duties, such as saving deployed artifacts in the artifacts files for later use, linking libraries, and so on. The `deployer` object is your main interface for the staging of deployment tasks.

Here are the methods of the deployer object. All the methods are synchronous:

- `deployer.deploy(contractAbstraction, args..., options)`: Deploys a specific contract specified by the contract abstraction object, with optional constructor arguments. This is useful for singleton contracts, so that only one instance of this contract exists for your DApp. This will set the address of the contract after deployment (that is, the `address` property in the artifacts file will equal the newly deployed address), and it will override any previous address stored. You can optionally pass an array of contracts, or an array of arrays, to speed up the deployment of multiple contracts. Additionally, the last argument is an optional object that can contain a single key, `overwrite`. If `overwrite` is set to `false`, the deployer won't deploy this contract if one has already been deployed. This method returns a promise.
- `deployer.link(library, destinations)`: Links an already deployed library to a contract or multiple contracts. The `destinations` argument can be a single contract abstraction or an array of multiple contract abstractions. If any contract within the destination doesn't rely on the library being linked, the deployer will ignore that contract. This method returns a promise.

- `deployer.then(function(){})`: This is used to run an arbitrary deployment step. Use it to call specific contract functions during your migration to add, edit, and reorganize contract data. Inside the callback function, you would use the contract abstraction APIs to deploy and link contracts.

It is possible to run the deployment steps conditionally based on the network being deployed to. To conditionally stage the deployment steps, write your migrations so that they accept a second parameter called `network`. One example use case can be that many of the popular libraries are already deployed to the main network; therefore, when using these networks, we will not deploy the libraries again and just link them instead. Here is a code example:

```
module.exports = function(deployer, network) {
  if (network != "live") {
   // Perform a different step otherwise.
  } else {
    // Do something specific to the network named "live".
  }
}
```

In the project, you will find two migration files, that is, `1_initial_migration.js` and `2_deploy_contracts.js`. The first file shouldn't be edited unless you know what you are doing. You are free to do anything with the other file. Here is the code for the `2_deploy_contracts.js` file:

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

module.exports = function(deployer) {
  deployer.deploy(ConvertLib);
  deployer.link(ConvertLib, MetaCoin);
  deployer.deploy(MetaCoin);
};
```

Here, we are creating abstractions for the `CovertLib` library and the `MetaCoin` contract at first. Regardless of which network is being used, we are deploying the `ConvertLib` library and then linking the library to the `MetaCoin` network and finally deploying the `MetaCoin` network.

To run the migrations, that is, to deploy the contracts, run this command:

```
truffle migrate --network development
```

Here, we are telling truffle to run migrations on the development network. If we don't provide the `--network` option, then it will use the network with the name `development` by default.

After you run the preceding command, you will notice that truffle will automatically update the `ConvertLib` library and `MetaCoin` contract addresses in the artifacts files and also update the `links`.

Here are some other important options you can provide to the migrate sub-command:

- `--reset`: Runs all migrations from the beginning instead of running from the last completed migration.
- `-f number`: Runs contracts from a specific migration.

You can find the address of the contracts and libraries of your project in various networks using the `truffle networks` command anytime.

# Unit testing contracts

Unit testing is a type of testing an app. It is a process in which the smallest testable parts of an application, called units, are individually and independently examined for proper operation. Unit testing can be done manually but is often automated.

Truffle comes with a unit testing framework by default to automate the testing of your contracts. It provides a clean room environment when running your test files; that is, truffle will rerun all of your migrations at the beginning of every test file to ensure you have a fresh set of contracts to test against.

Truffle lets you write simple and manageable tests in two different ways:

- In JavaScript, to exercise your contracts from the app client
- In Solidity, to exercise your contracts from other contracts

Both styles of tests have their advantages and drawbacks. We will learn both ways of writing tests.

All test files should be located in the `./test` directory. Truffle will run test files only with these file extensions: `.js`, `.es`, `.es6`, and `.jsx`, and `.sol`. All other files are ignored.

The ethereumjs-testrpc is significantly faster than other clients when running automated tests. Moreover, testrpc contains special features that truffle takes advantage of to speed up the test runtime by almost 90 percent. As a general workflow, we recommend that you use testrpc during normal development and testing and then run your tests once against go-ethereum or another official ethereum client when you're gearing up to deploy to live or production networks.

## Writing tests in JavaScript

Truffle's JavaScript testing framework is built on top of mocha. Mocha is a JavaScript framework to write tests, whereas chai is an assertion library.

Testing frameworks are used to organize and execute tests, whereas assertion libraries provide utilities to verify that things are correct. Assertion libraries make it a lot easier to test your code so you don't have to perform thousands of if statements. Most of the testing frameworks don't have an assertion library included and let the user plug which one they want to use.

Before continuing further, you need to learn how to write tests with mocha and chai. To learn mocha, visit `https://mochajs.org/` and to learn chai, visit `http://chaijs.com/`.

Your tests should exist in the `./test` directory, and they should end with a `.js` extension.

Contract abstractions are the basis for making contract interaction possible from JavaScript. Because truffle has no way of detecting which contracts you'll need to interact with within your tests, you'll need to ask for these contracts explicitly. You do this by using the `artifacts.require()` method. So the first thing that should be done in test files is to create abstractions for the contracts that you want to test.

Then, the actual tests should be written. Structurally, your tests should remain largely unchanged from those of mocha. The test files should contain code that mocha will recognize as an automated test. What makes truffle tests different from mocha is the `contract()` function: this function works exactly like `describe()`, except that it signals truffle to run all migrations. The `contract()` function works like this:

- Before each `contract()` function is run, your contracts are redeployed to the running ethereum node, so the tests within it run with a clean contract state
- The `contract()` function provides a list of accounts made available by your ethereum node, which you can use to write tests

Since truffle uses mocha under the hood, you can still use `describe()` to run normal mocha tests whenever truffle features are unnecessary.

Here is the default test code generated by truffle to test the `MetaCoin` contract. You will find this code in the `metacoin.js` file:

```
// Specifically request an abstraction for MetaCoin.sol
var MetaCoin = artifacts.require("./MetaCoin.sol");

contract('MetaCoin', function(accounts) {
  it("should put 10000 MetaCoin in the first account", function() {
    return MetaCoin.deployed().then(function(instance) {
      return instance.getBalance.call(accounts[0]);
    }).then(function(balance) {
      assert.equal(balance.valueOf(), 10000, "10000 wasn't in the first
account");
    });
  });
  it("should send coin correctly", function() {
    var meta;

    // Get initial balances of first and second account.
    var account_one = accounts[0];
    var account_two = accounts[1];

    var account_one_starting_balance;
    var account_two_starting_balance;
    var account_one_ending_balance;
    var account_two_ending_balance;

    var amount = 10;

    return MetaCoin.deployed().then(function(instance) {
      meta = instance;
      return meta.getBalance.call(account_one);
    }).then(function(balance) {
      account_one_starting_balance = balance.toNumber();
      return meta.getBalance.call(account_two);
    }).then(function(balance) {
      account_two_starting_balance = balance.toNumber();
      return meta.sendCoin(account_two, amount, {from: account_one});
    }).then(function() {
      return meta.getBalance.call(account_one);
    }).then(function(balance) {
      account_one_ending_balance = balance.toNumber();
```

```
        return meta.getBalance.call(account_two);
      }).then(function(balance) {
        account_two_ending_balance = balance.toNumber();

        assert.equal(account_one_ending_balance, account_one_starting_balance
- amount, "Amount wasn't correctly taken from the sender");
        assert.equal(account_two_ending_balance, account_two_starting_balance
+ amount, "Amount wasn't correctly sent to the receiver");
      });
    });
});
```

In the preceding code, you can see that all the contract's interaction code is written using the `truffle-contract` library. The code is self-explanatory.

Finally, truffle gives you access to mocha's configuration so you can change how mocha behaves. mocha's configuration is placed under a `mocha` property in the `truffle.js` file's exported object. For example, take a look at this:

```
mocha: {
  useColors: true
}
```

## Writing tests in Solidity

Solidity test code is put in `.sol` files. Here are the things you need to note about Solidity tests before writing tests using Solidity:

- Solidity tests shouldn't extend from any contract. This makes your tests as minimal as possible and gives you complete control over the contracts you write.
- Truffle provides a default assertion library for you, but you can change this library at any time to fit your needs.
- You should be able to run your Solidity tests against any ethereum client.

To learn how to write tests in Solidity, let's explore the default Solidity test code generated by truffle. This is the code, and it can be found in the `TestMetacoin.sol` file:

```
pragma Solidity ^0.4.2;

import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/MetaCoin.sol";

contract TestMetacoin {
  function testInitialBalanceUsingDeployedContract() {
    MetaCoin meta = MetaCoin(DeployedAddresses.MetaCoin());

    uint expected = 10000;

    Assert.equal(meta.getBalance(tx.origin), expected, "Owner should have
10000 MetaCoin initially");
  }

  function testInitialBalanceWithNewMetaCoin() {
    MetaCoin meta = new MetaCoin();

    uint expected = 10000;

    Assert.equal(meta.getBalance(tx.origin), expected, "Owner should have
10000 MetaCoin initially");
  }

}
```

Here is how the preceding code works:

- Assertion functions such as `Assert.equal()` are provided to you by the `truffle/Assert.sol` library. This is the default assertion library; however, you can include your own assertion library as long as the library loosely integrates with truffle's test runner by triggering the correct assertion events. Assertion functions fire events, which are caught by truffle, and information is displayed. This is the architecture of Solidity assertion libraries in truffle. You can find all the available assertion functions in `Assert.sol` (`https://github.com/ConsenSys/truffle/blob/beta/lib/testing/Assert.sol`).
- In the import path, `truffle/Assert.sol`, `truffle` is the package name. We will learn more about packages later.
- The addresses of your deployed contracts (that is, contracts that were deployed as part of your migrations) are available through the `truffle/DeployedAddresses.sol` library. This is provided by truffle and is recompiled and relinked before each test suite is run. This library provides functions for all of your deployed contracts in the form of `DeployedAddresses.<contract name>()`. This will return an address that you can then use to access that contract.

- In order to use the deployed contract, you'll have to import the contract code into your test suite. Notice `import "../contracts/MetaCoin.sol";` in the preceding example. This import is relative to the test contract, which exists in the `./test` directory, and it goes outside of the test directory in order to find the `MetaCoin` contract. It then uses that contract to cast the address to the `MetaCoin` type.
- All test contracts must start with `Test`, using an uppercase T. This distinguishes this contract from test helpers and project contracts (that is, the contracts under test), letting the test runner know which contracts represent test suites.
- Like test contract names, all test functions must start with `test`, in lowercase. Each test function is executed as a single transaction in order of appearance in the test file (such as your JavaScript tests). Assertion functions provided by `truffle/Assert.sol` trigger events that the test runner evaluates to determine the result of the test. Assertion functions return a Boolean that represents the outcome of the assertion, which you can use to return from the test early to prevent execution errors (that is, errors that `testrpc` will expose).
- You are provided with many test hooks, shown in the following example. These hooks are `beforeAll`, `beforeEach`, `afterAll`, and `afterEach`, which are the same hooks provided by mocha in your JavaScript tests. You can use these hooks to perform setup and teardown actions before and after each test or before and after each suite is run. Like test functions, each hook is executed as a single transaction. Note that some complex tests will need to perform a significant amount of setup that might overflow the gas limit of a single transaction; you can get around this limitation by creating many hooks with different suffixes, as shown in the following example:

```
import "truffle/Assert.sol";

contract TestHooks {
  uint someValue;

  function beforeEach() {
    someValue = 5;
  }

  function beforeEachAgain() {
    someValue += 1;
  }

  function testSomeValueIsSix() {
    uint expected = 6;

    Assert.equal(someValue, expected, "someValue should have been 6");
  }
}
```

- This test contract also shows that your `test` functions and `hook` functions all share the same contract state. You can set up the contract data before the test, use that data during the test, and reset it afterward in preparation for the next one. Note that just like your JavaScript tests, your next test function will continue from the state of the previous test function that ran.

Truffle doesn't provide a direct way to test whether your contract should and shouldn't throw exception (that is, for contracts that use throw to signify an expected error). But a hacky solution is there for this, which you can find at `http://truffleframework.com/tutorials/testing-for-throws-in-Solidity-tests`.

## How to send ether to a test contract

To send ether to your Solidity test contract, it should have a public function that returns `uint`, called `initialBalance` in that contract. This can be written directly as a function or a public variable. When your test contract is deployed to the network, truffle will send that amount of ether from your test account to your test contract. Your test contract can then use that ether to script ether interactions within your contract under test. Note that `initialBalance` is optional and not required. For example, take a look at the following code:

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

contract TestContract {
  // Truffle will send the TestContract one Ether after deploying the
contract.
  public uint initialBalance = 1 ether;

  function testInitialBalanceUsingDeployedContract() {
    MyContract myContract = MyContract(DeployedAddresses.MyContract());

    // perform an action which sends value to myContract, then assert.
    myContract.send(...);
  }

  function () {
    // This will NOT be executed when Ether is sent. o/
  }
}
```

Truffle sends ether to your test contract in a way that does not execute a fallback function, so you can still use the fallback function within your Solidity tests for advanced test cases.

## Running tests

To run your test scripts, just run this command:

```
truffle test
```

Alternatively, you can specify a path to a specific file you want to run. For example, take a look at this:

```
truffle test ./path/to/test/file.js
```

# Package management

A truffle package is a collection of smart contracts and their artifacts. A package can depend on zero or more packages, that is, you use the package's smart contracts and artifacts. When using a package within your own project, it is important to note that there are two places where you will be using the package's contracts and artifacts: within your project's contracts and within your project's JavaScript code (migrations and tests).

Projects created with truffle have a specific layout by default, which enables them to be used as packages. The most important directories in a `truffle` package are the following:

- `/contracts`
- `/build/contracts` (created by truffle)

The first directory is your contracts directory, which includes your raw Solidity contracts. The second directory is the `/build/contracts` directory, which holds build artifacts in the form of `.json` files.

Truffle supports two kinds of package builds: `npm` and `ethpm` packages. You must know what `npm` packages are, but let's look at what `ethpm` packages are. `Ethpm` is a package registry for ethereum. You can find all `ethpm` packages at `https://www.ethpm.com/`. It follows the ERC190 (`https://github.com/ethereum/EIPs/issues/190`) spec for publishing and consuming smart contract packages.

## Package management via NPM

Truffle comes with npm integration by default and is aware of the `node_modules` directory in your project, if it exists. This means that you can use and distribute contracts or libraries via npm, making your code available to others and other's code available to you. You can also have a `package.json` file in your project. You can simply install any `npm` package in your project and import it in any of the JavaScript files, but it would be called a truffle package only if it contains the two directories mentioned earlier. Installing an `npm` package in a truffle project is the same as installing an `npm` package in any `Node.js` app.

## Package management via EthPM

When installing EthPM packages, an `installed_contracts` directory is created if it doesn't exist. This directory can be treated in a manner similar to the `node_modules` directory.

Installing a package from EthPM is nearly as easy as installing a package via NPM. You can simply run the following command:

```
truffle install <package name>
```

You can also install a package at a specific version:

```
truffle install <package name>@<version>
```

Like NPM, EthPM versions follow semver. Your project can also define an `ethpm.json` file, which is similar to `package.json` for npm packages. To install all dependencies listed in the `ethpm.json` file, run the following:

```
truffle install
```

An example `ethpm.json` file looks like this:

```
{
  "package_name": "adder",
  "version": "0.0.3",
  "description": "Simple contract to add two numbers",
  "authors": [
    "Tim Coulter <tim.coulter@consensys.net>"
  ],
  "keywords": [
    "ethereum",
    "addition"
  ],
  "dependencies": {
    "owned": "^0.0.1"
  },
  "license": "MIT"
}
```

Creating and publishing an `npm` package for truffle is the same process as creating any other `npm` package. To learn how to create and publish an `ethpm` package, visit `http://truffleframework.com/docs/getting_started/packages-ethpm#publishing-your-own-package`. Regardless of whether you are publishing your package as an `npm` package or `ethpm` package, you need to run the `truffle networks --clean` command. When this command is run, it deletes artifacts for all those networks IDs that match only the `*` wildcard character in the configuration file. This is done as these addresses will be invalid for the other projects consuming this package, as these networks are most likely to be private as they are used for development purpose only. You shouldn't omit this command unless you know what you are doing. It will fail to delete any artifacts for private networks listed as a constant, so you need to delete them manually.

## Using contracts of packages within your contracts

To use a package's contracts within your contracts, it can be as simple as Solidity's `import` statement. When your `import` path isn't explicitly relative or absolute, it signifies to truffle that you're looking for a file from a specific named package. Consider this example using the `example-truffle-library` (`https://github.com/ConsenSys/example-truffle-library`):

```
import "example-truffle-library/contracts/SimpleNameRegistry.sol";
```

Since the path didn't start with `./`, truffle knows to look in your project's `node_modules` or `installed_contracts` directory for the `example-truffle-library` folder. From there, it resolves the path to provide you with the contract you requested.

## Using artifacts of packages within your JavaScript code

To interact with a package's artifacts within JavaScript code, you simply need to require that package's `.json` files and then use `truffle-contract` to turn them into usable abstractions:

```
var contract = require("truffle-contract");
var data = require("example-truffle-
library/build/contracts/SimpleNameRegistry.json");
var SimpleNameRegistry = contract(data);
```

## Accessing a package's contracts deployed addresses in Solidity

Sometimes, you may want your contracts to interact with the package's previously deployed contracts. Since the deployed addresses exist within the package's `.json` files, Solidity code cannot directly read contents of these files. So, the flow of making Solidity code access the addresses in `.json` files is by defining functions in Solidity code to set dependent contract addresses, and after the contract is deployed, call those functions using JavaScript to set the dependent contract addresses.

So you can define your contract code like this:

```
import "example-truffle-library/contracts/SimpleNameRegistry.sol";

contract MyContract {
  SimpleNameRegistry registry;
  address public owner;

  function MyContract {
    owner = msg.sender;
  }

  // Simple example that uses the deployed registry from the package.
  function getModule(bytes32 name) returns (address) {
    return registry.names(name);
  }

  // Set the registry if you're the owner.
  function setRegistry(address addr) {
    if (msg.sender != owner) throw;

    registry = SimpleNameRegistry(addr);
  }
}
```

This is what your migration should look like:

```
var SimpleNameRegistry = artifacts.require("example-truffle-
library/contracts/SimpleNameRegistry.sol");

module.exports = function(deployer) {
  // Deploy our contract, then set the address of the registry.
  deployer.deploy(MyContract).then(function() {
    return MyContract.deployed();
  }).then(function(deployed) {
    return deployed.setRegistry(SimpleNameRegistry.address);
  });
};
```

# Using truffle's console

Sometimes, it's nice to work with your contracts interactively for testing and debugging purposes or to execute transactions by hand. Truffle provides you with an easy way to do this via an interactive console, with your contracts available and ready to use.

To open the console, run this command:

```
truffle console
```

The console connects to an ethereum node based on your project configuration. The preceding command also takes a `--network` option to specify a specific node to connect to.

Here are the features of the console:

- You can run the command in the console. For instance, you can type `migrate --reset` within the console, and it will be interpreted the same as if you ran `truffle migrate --reset` from outside the console.
- All of your compiled contracts are available and ready for use.
- After each command (such as `migrate --reset`), your contracts are re-provisioned, so you can start using the newly assigned addresses and binaries immediately.
- The `web3` object is made available and is set to connect to your ethereum node.
- All commands that return a promise will automatically be resolved and the result printed, removing the need to use `.then()` for simple commands. For example, you can write code like this:
  `MyContract.at("0xabcd...").getValue.call();`

## Running external scripts in truffle's context

Often, you may want to run external scripts that interact with your contracts. Truffle provides an easy way to do this, bootstrapping your contracts based on your desired network and connecting to your ethereum node automatically as per your project configuration.

To run an external script, run this command:

```
truffle exec <path/to/file.js>
```

In order for external scripts to be run correctly, truffle expects them to export a function that takes a single parameter as a callback. You can do anything you'd like within this script as long as the callback is called when the script finishes. The callback accepts an error as its first and only parameter. If an error is provided, execution will halt and the process will return a nonzero exit code.

This is the structure external scripts must follow:

```
module.exports = function(callback) {
  // perform actions
  callback();
}
```

## Truffle's build pipeline

Now that you know how to compile, deploy, and test smart contracts using truffle, it's time to build a client for our altcoin. Before we get into how to build a client using truffle, you need to know that it doesn't allow us to sign transactions using accounts stored outside of the ethereum node; that is, it doesn't have anything similar to `sendRawTransaction` and the reasons are the same as those for `truffle-contract`.

Building a client using truffle means first integrating truffle's artifacts in your client source code and then preparing the client's source code for deployment.

To build a client, you need to run this command:

```
truffle build
```

When this command is run, truffle will check how to build the client by inspecting the `build` property in the project's configuration file.

## Running an external command

A command-line tool can be used to build a client. When the `build` property is a string, truffle assumes that we want to run a command to build the client, so it runs the string as a command. The command is given ample environment variables with which to integrate with truffle.

You can make truffle run a command-line tool to build the client using similar configuration code:

```
module.exports = {
  // This will run the &grave;webpack&grave; command on each build.
  //
  // The following environment variables will be set when running the
command:
  // WORKING_DIRECTORY: root location of the project
  // BUILD_DESTINATION_DIRECTORY: expected destination of built assets
  // BUILD_CONTRACTS_DIRECTORY: root location of your build contract files
(.sol.js)
  //
  build: "webpack"
}
```

## Running a custom function

A JavaScript function can be used to build a client. When the `build` property is a function, truffle will run that function whenever we want to build the client. The function is given a lot of information about the project with which to integrate with truffle.

You can make truffle run a function to build the client using similar configuration code:

```
module.exports = {
  build: function(options, callback) {
     // Do something when a build is required. &grave;options&grave;
contains these values:
     //
     // working_directory: root location of the project
     // contracts_directory: root directory of .sol files
     // destination_directory: directory where truffle expects the built
assets (important for &grave;truffle serve&grave;)
  }
}
```

You could also create an object, which contains a `build` method like the one here. This is great for those who want to publish a package to build a client.

## Truffle's default builder

Truffle provides the `truffle-default-builder npm` package, which is termed the default builder for truffle. This builder exports an object, which has a build method, which works exactly like the previously mentioned method.

The default builder can be used to build a web client for your DApp, whose server only serves static files, and all the functionality is on the frontend.

Before we get further into how to use the default builder, first install it using this command:

```
npm install truffle-default-builder --save
```

Now change your configuration file to this:

```
var DefaultBuilder = require("truffle-default-builder");

module.exports = {
  networks: {
    development: {
      host: "localhost",
      port: 8545,
      network_id: "10"
    },
    live: {
         host: "localhost",
      port: 8545,
      network_id: "1"
    }
  },
  build: new DefaultBuilder({
    "index.html": "index.html",
    "app.js": [
      "javascripts/index.js"
    ],
    "bootstrap.min.css": "stylesheets/bootstrap.min.css"
  })
};
```

The default builder gives you complete control over how you want to organize the files and folders of your client.

This configuration describes `targets` (left-hand side) with files, folders, and arrays of files that make up the `targets` contents (right-hand side). Each target will be produced by processing the files on the right-hand side based on their file extension, concatenating the results together, and then saving the resultant file (the target) into the build destination. Here, a string is specified on the right-hand side instead of an array, and that file will be processed, if needed, and then copied over directly. If the string ends in a `"/"`, it will be interpreted as a directory and the directory will be copied over without further processing. All paths specified on the right-hand side are relative to the `app/` directory.

You can change this configuration and directory structure at any time. You aren't required to have a `javascripts` and `stylesheets` directory, for example, but make sure you edit your configuration accordingly.

If you want the default builder to integrate truffle on the frontend of your web application, make sure you have a build target called `app.js`, which the default builder can append code to. It will not integrate truffle with any other filename.

Here are the features of the default builder:

- Automatically imports your compiled contract artifacts, deployed contract information, and ethereum node configuration into the client source code
- Includes recommended dependencies, including web3 and `truffle-contract`
- Compiles `ES6` and `JSX` files
- Compiles `SASS` files
- Minifies `asset` files

You can use the `truffle watch` command, which watches for changes in the `contracts` directory, the `app` directory, and the configuration file. When there's a change, it recompiles the contracts and generates new artifact files and then rebuilds the client. But it doesn't run migrations and tests.

## Building a client

Now let's write a client for our DApp and build it using truffle's default builder. First of all, create files and directories based on the preceding configuration we set: create an `app` directory and inside it, create an `index.html` file and two directories called `javascripts` and `styelsheets`. Inside the `javascripts` directory, create a file called `index.js` and in the `stylesheets` directory, download and place the CSS file of Bootstrap 4. You can find it at `https://v4-alpha.getbootstrap.com/getting-started/download/#bootstrap-css-and-js`.

In the `index.html` file, place this code:

```
<!doctype html>
<html>
   <head>
         <link rel="stylesheet" type="text/css" href="bootstrap.min.css">
   </head>
   <body>
   <div class="container">
         <div class="row">
         <div class="col-md-6">
         <br>
         <h2>Send Metacoins</h2>
         <hr>
         <form id="sendForm">
         <div class="form-group">
               <label for="fromAddress">Select Account Address</label>
         <select class="form-control" id="fromAddress">
         </select>
         </div>
         <div class="form-group">
         <label for="amount">How much metacoin do you want to send?
         </label>
         <input type="text" class="form-control" id="amount">
         </div>
         <div class="form-group">
         <label for="toAddress">Enter the address to which you want to
           send matacoins</label>
         <input type="text" class="form-control" id="toAddress"
             placeholder="Prefixed with 0x">
         </div>
         <button type="submit" class="btn btn-primary">Submit</button>
         </form>
         </div>
         <div class="col-md-6">
         <br>
         <h2>Find Balance</h2>
         <hr>
         <form id="findBalanceForm">
         <div class="form-group">
               <label for="address">Select Account Address</label>
               <select class="form-control" id="address">
               </select>
         </div>
         <button type="submit" class="btn btn-primary">Check
            Balance</button>
         </form>
         </div>
         </div>
         </div>
         <script type="text/javascript" src="/app.js"></script>
   </body>
</html>


<!doctype html>
 <html>
     <head>
         <link rel="stylesheet" type="text/css" href="bootstrap.min.css">
     </head>
     <body>
         <div class="container">
             <div class="row">
                 <div class="col-md-6">
                     <br>
                     <h2>Send Metacoins</h2>
                     <hr>
                     <form id="sendForm">
                         <div class="form-group">
                             <label for="fromAddress">Select Account
Address</label>
                             <select class="form-control" id="fromAddress">
                             </select>
                           </div>
                           <div class="form-group">
                             <label for="amount">How much metacoin you want
to send?</label>
                             <input type="text" class="form-control"
id="amount">
                           </div>
                           <div class="form-group">
                             <label for="toAddress">Enter the address to
which you want to send matacoins</label>
                             <input type="text" class="form-control"
```

```
id="toAddress" placeholder="Prefixed with 0x">
                           </div>
                           <button type="submit" class="btn btn-
primary">Submit</button>
                     </form>
                 </div>
                 <div class="col-md-6">
                     <br>
                     <h2>Find Balance</h2>
                     <hr>
                     <form id="findBalanceForm">
                         <div class="form-group">
                             <label for="address">Select Account
Address</label>
                             <select class="form-control" id="address">
                             </select>
                           </div>
                           <button type="submit" class="btn btn-
primary">Check Balance</button>
                     </form>
                 </div>
             </div>
         </div>
         <script type="text/javascript" src="/app.js"></script>
     </body>
 </html>
```

In the preceding code, we are loading the `bootstrap.min.css` and `app.js` files. We have two forms: one is to send metacoins to a different account and the other one is to check the metacoins balance of an account. In the first form, the user has to select an account and then enter the amount of metacoin to send and the address that it wants to send to. And in the second form, the user simply has to select the address whose metacoin balance it wants to check.

In the `index.js` file, place this code:

```
window.addEventListener("load", function(){
   var accounts = web3.eth.accounts;

   var html = "";

   for(var count = 0; count < accounts.length; count++)
   {
         html = html + "<option>" + accounts[count] + "</option>";
   }

   document.getElementById("fromAddress").innerHTML = html;
```

```
      document.getElementById("address").innerHTML = html;

      MetaCoin.detectNetwork();
   })

   document.getElementById("sendForm").addEventListener("submit", function(e){
      e.preventDefault();

      MetaCoin.deployed().then(function(instance){
            return
   instance.sendCoin(document.getElementById("toAddress").value,
   document.getElementById("amount").value, {
                  from:
   document.getElementById("fromAddress").options[document.getElementById("fro
   mAddress").selectedIndex].value
            });
      }).then(function(result){
            alert("Transaction mined successfully. Txn Hash: " + result.tx);
      }).catch(function(e){
            alert("An error occured");
      })
   })

   document.getElementById("findBalanceForm").addEventListener("submit",
   function(e){
      e.preventDefault();

      MetaCoin.deployed().then(function(instance){
            return
   instance.getBalance.call(document.getElementById("address").value);
      }).then(function(result){
            console.log(result);
            alert("Balance is: " + result.toString() + " metacoins");
      }).catch(function(e){
            alert("An error occured");
      })
   })
```

Here is how the code works:

1. The `truffle-default-builder` makes artifacts objects available under the `__contracts__` global object.
2. It also makes available contract abstractions for all the contracts available as global variables with the variable name the same as the contract name.

3. It also provides the web3 object by already setting the provider. It also sets the provider for the contract abstractions. It makes the web3 object connect to the network with the name `development` and if it doesn't exist, then the default value is `http://localhost:8545`.
4. In the preceding code, at first, we wait for the page to load, and once loaded, we retrieve the list of accounts in the connected node and display them in both the forms. And we call the `detectNetwork()` method of the `MetaCoin` abstraction as well.
5. Then, we have `submit` event handlers for both the forms. They both do what they are supposed to do and display the result in a popup.
6. When the first form is submitted, we get the `MetaCoin` contract's deployed instance and call the `sendCoin` method with the correct arguments.
7. When the second form is submitted, we retrieve the balance of the selected account by calling the `getBalance` method in the EVM instead of broadcasting a transaction.

Now go ahead and run the truffle build command, and you will notice that truffle will create `index.html`, `app.js`, and `bootstrap.min.css` files in the `build` directory and put the client's final deployment code in them.

# Truffle's server

Truffle comes with an in-built web server. This web server simply serves the files in the `build` directory with a proper MIME type set. Apart from this, it's not configured to do anything else.

To run the web server, run this command:

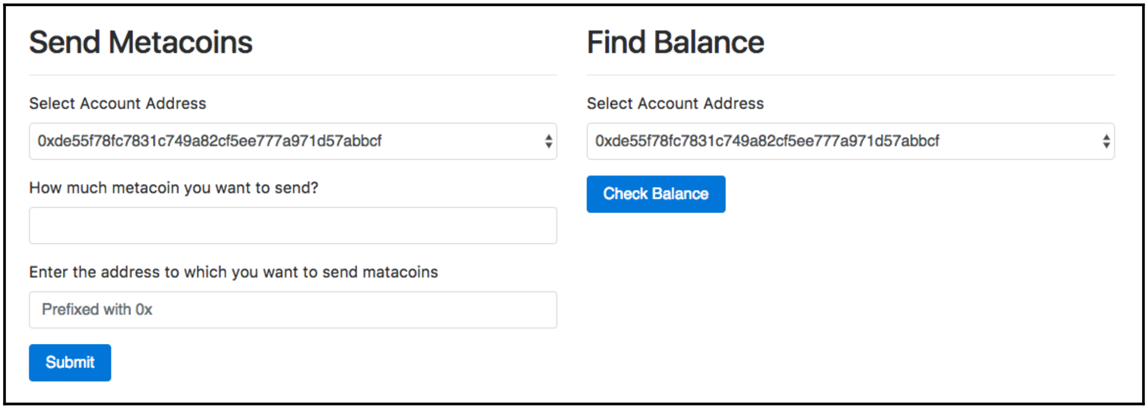
```
truffle serve
```

The server runs on port number 8080 by default. But you can use the `-p` option to specify a different port number.

Similar to truffle watch, this web server also watches for changes in the `contracts` directory, the `app` directory, and the configuration file. When there's a change, it recompiles the contracts and generates new artifacts files and then rebuilds the client. But it doesn't run migrations and tests.

As the truffle-default-builder places the final deployable code in the build directory, you can simply run `truffle serve` to serve the files via the Web.

Let's test our web client. Visit `http://localhost:8080`, and you will see this screenshot:



Send Metacoins

Select Account Address

0xde55f78fc7831c749a82cf5ee777a971d57abbcf

How much metacoin you want to send?

Enter the address to which you want to send matacoins

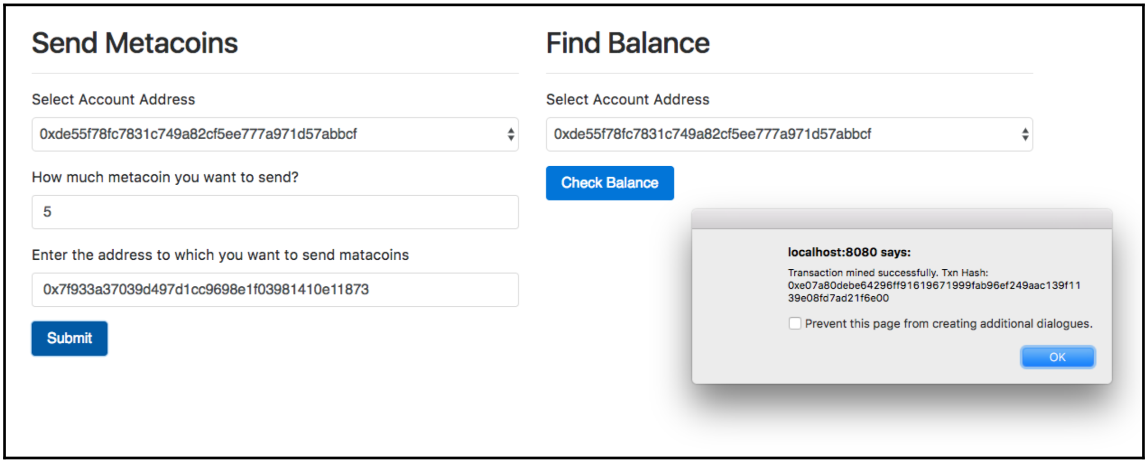
Prefixed with 0x

Find Balance

Select Account Address

0xde55f78fc7831c749a82cf5ee777a971d57abbcf

The account addresses in the selected boxes will differ for you. Now at the time of deploying the contract, the contract assigns all the metacoins to the address that deploys the contract; so here, the first account will have a balance of 10,000 metacoins. Now send five metacoins from the first account to the second account and click on **Submit**. You will see a screen similar to what is shown in the following screenshot:



Send Metacoins

Select Account Address

0xde55f78fc7831c749a82cf5ee777a971d57abbcf

How much metacoin you want to send?

5

Enter the address to which you want to send matacoins

0x7f933a37039d497d1cc9698e1f03981410e11873

Find Balance

Select Account Address

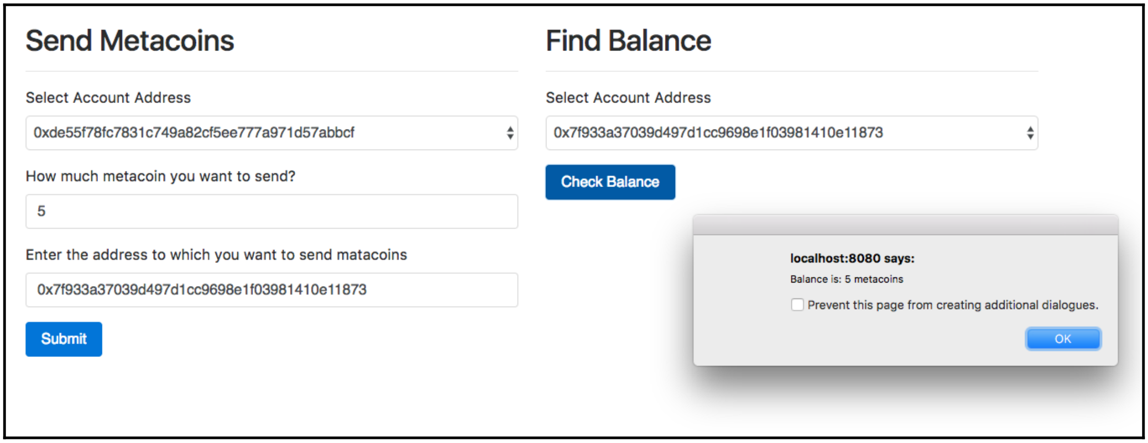
0xde55f78fc7831c749a82cf5ee777a971d57abbcf

**localhost:8080 says:**

Transaction mined successfully. Txn Hash:
0xe07a80debe64296ff91619671999fab96ef249aac139f11
39e08fd7ad21f6e00

□ Prevent this page from creating additional dialogues.

Now check the balance of the second account by selecting the second account in the select box of the second form and then click on the **Check Balance** button. You will see a screen similar to what is shown in the following screenshot:



Send Metacoins

Select Account Address

0xde55f78fc7831c749a82cf5ee777a971d57abbcf

How much metacoin you want to send?

5

Enter the address to which you want to send matacoins

0x7f933a37039d497d1cc9698e1f03981410e11873

Find Balance

Select Account Address

0x7f933a37039d497d1cc9698e1f03981410e11873

localhost:8080 says:

Balance is: 5 metacoins

□ Prevent this page from creating additional dialogues.

# Summary

In this chapter, we learned in depth how to build DApps and their respective clients using truffle. We look at how truffle makes it really easy to write, compile, deploy, and test DApps. We also saw how easy it is to switch between networks in clients using `truffle-contract` without touching the source code. Now you are ready to start building enterprise-level DApps using truffle.

In the next chapter, we will build a decentralized alarm clock app that pays you to wake up on time using the truffle and ethereum alarm clock DApp.

Just replace the `i.e.` with a colon `":"`.

# 9
# Building a Consortium Blockchain

Consortiums (an association, typically of several participants such as banks, e-commerce sites, government entities, hospitals, and so on) can use blockchain technology to solve many problems and make things faster and cheaper. Although they figure out how blockchain can help them, an Ethereum implementation of blockchain doesn't specifically fit them for all cases. Although there are other implementations of blockchain (for example, Hyperledger) that are built specially for consortium, as we learned Ethereum throughout the book, we will see how we can hack Ethereum to build a consortium blockchain. Basically, we will be using parity to build a consortium blockchain. Although there are other alternatives to parity, such as J.P. Morgan's quorum, we will use parity as at the time of writing this book, it has been in existence for some time, and many enterprises are already using it, whereas other alternatives are yet to be used by any enterprises. But for your requirements, parity may not be the best solution; therefore, investigate all the others too before deciding which one to use.

In this chapter, we'll cover the following topics:

- Why is Ethereum unfit for consortium blockchain?
- What is parity node and what are its features?
- What is the Proof-of-Authority consensus protocol and what types of PoA are supported by parity?
- How does the Aura consensus protocol work?
- Downloading and installing parity
- Building a consortium blockchain using parity

# What is a consortium blockchain?

To understand what a consortium blockchain is, or, in other words, what kind of blockchain implementation consortiums need, let's check out an example. Banks want to build a blockchain to make money transfers easier, faster, and cheaper. In this case, here are the things they need:

1. **Speed**: They need a blockchain network that can confirm transactions in near-real time. Currently, the Ethereum blockchain network block time is 12 seconds, and clients usually wait for a couple of minutes before confirming a transaction.
2. **Permissioned**: They want the blockchain to be permissioned. Permissioning itself means various different things. For example: permissioning can include taking permission to join the network, it can include taking permission to to be able create blocks, it can also be taking permission to be able to send specific transactions and so on.
3. **Security**: PoW isn't secure enough for private networks as there is a limited number of participants; therefore, there isn't enough hash power produced to make it secure. So, there is a need for a consensus protocol that can keep the blockchain secure and immutable.
4. **Privacy**: Although the network is private, there is still a need for privacy in the network itself. There are two kinds of privacy:
5. **Identity privacy**: Identity privacy is the act of making the identity untraceable. The solution we saw earlier to gain identity privacy was to use multiple Ethereum account addresses. But if multiple Ethereum accounts are used, then smart contracts will fail ownership validation as there is no way to know whether all of these accounts actually belong to the same user.
6. **Data privacy:** Sometimes, we don't want the data to be visible to all the nodes in the network, but to specific nodes only.

Overall, in this chapter, we will learn how to solve these issues in Ethereum.

# What is Proof-of-Authority consensus?

PoA is a consensus mechanism for blockchain in which consensus is achieved by referring to a list of validators (referred to as authorities when they are linked to physical entities). Validators are a group of accounts/nodes that are allowed to participate in the consensus; they validate the transactions and blocks.

Unlike PoW or PoS, there is no mining mechanism involved. There are various types of PoA protocols, and they vary depending on how they actually work. Hyperledger and Ripple are based on PoA. Hyperledger is based on PBFT, whereas ripple uses an iterative process.

# Introduction to parity

Parity is an Ethereum node written from the ground up for correctness/verifiability, modularization, low footprint, and high performance. It is written in Rust programming language, a hybrid imperative/OO/functional language with an emphasis on efficiency. It is professionally developed by Parity Technologies. At the time of writing this book, the latest version of parity is 1.7.0, and we will be using this version. We will learn as much as is required to build a consortium blockchain. To learn parity in depth, you can refer to the official documentation.

It has a lot more features than go-ethereum, such as web3 dapp browser, much more advanced account management, and so on. But what makes it special is that it supports **Proof-of-Authority** (**PoA**) along with PoW. Parity currently su+-pports Aura and Tendermint PoA protocols. In future, it may support some more PoA protocols. Currently, parity recommends the use of Aura instead of Tendermint as Tendermint is still under development.

Aura is a much better solution for permissioned blockchains than PoW as it has better block time and provides much better security in private networks.

# Understanding how Aura works

Let's see at a high level how Aura works. Aura requires the same list of validators to be specified in each node. This is a list of account addresses that participate in the consensus. A node may or may not be a validating node. Even a validating node needs to have this list so that it can itself reach a consensus.

This list can either be provided as a static list in the genesis file if the list of validators is going to remain the same forever, or be provided in a smart contract so that it can be dynamically updated and every node knows about it. In a smart contract, you can configure various strategies regarding who can add new validators.

The block time is configurable in the genesis file. It's up to you to decide the block time. In private networks, a block time as low as three seconds works well. In Aura, after every three seconds, one of the validators is selected and this validator is responsible for creating, verifying, signing, and broadcasting the block. We don't need to understand much about the actual selection algorithm as it won't impact our dapp development. But this is the formula to calculate the next validator, `(UNIX_TIMESTAMP / BLOCK_TIME % NUMBER_OF_TOTAL_VALIDATORS)`. The selection algorithm is smart enough to give equal chances to everyone When other nodes receive a block, they check whether it's from the next valid validator or not; and if not, they reject it. Unlike PoW, when a validator creates a block, it is not rewarded with ether. In Aura, it's up to us to decide whether to generate empty blocks or not when there are no transactions.

You must be wondering what will happen if the next validator node, due to some reason, fails to create and broadcast the next block. To understand this, let's take an example: suppose A is the validator for the next block, which is the fifth block, and B is the validator for the sixth block. Assume block time is five seconds. If A fails to broadcast a block, then after five seconds when B's turn arrives, it will broadcast a block. So nothing serious happens actually. The block timestamp will reveal these details.

You might also be wondering whether there are chances of networks ending up with multiple different blockchains as it happens in PoW when two miners mine at the same time. Yes, there are many ways this can happen. Let's take an example and understand one way in which this can happen and how the network resolves it automatically. Suppose there are five validators: A, B, C, D, and E. Block time is five seconds. Suppose A is selected first and it broadcasts a block, but the block doesn't reach D and E due to some reason; so they will think A didn't broadcast the block. Now suppose the selection algorithm selects B to generate the next block; then B will generate the next block on top of A's block and broadcast to all the nodes. D and E will reject it because the previous block hash will not match. Due to this, D and E will form a different chain, and A, B, and C will form a different chain. A, B, and C will reject blocks from D and E, and D and E will reject blocks from A, B, and C. This issue is resolved among the nodes as the blockchain that is with A, B and C is more accurate than the blockchain with D and E; therefore D and E will replace their version of blockchain with the blockchain held with A, B, and C. Both these versions of the blockchain will have different accuracy scores, and the score of the first blockchain will be more than the second one. When B broadcasts its block, it will also provide the score of its blockchain, and as its score is higher, D and E will have replaced their blockchain with B's blockchain. This is how conflicts are resolved. The chain score of blockchain is calculated using `(U128_max * BLOCK_NUMBER_OF_LATEST_BLOCK - (UNIX_TIMESTAMP_OF_LATEST_BLOCK / BLOCK_TIME))`. Chains are scored first by their length (the more blocks, the better). For chains of equal length, the chain whose last block is older is chosen.

You can learn more about Aura in depth at `https://github.com/paritytech/parity/wiki/Aura`.

# Getting parity running

Parity requires Rust version 1.16.0 to build. It is recommended to install Rust through rustup.

## Installing rust

If you don't already have rustup, you can install it like this.

### Linux

On Linux-based operating systems, run this command:

```
curl https://sh.rustup.rs -sSf | sh
```

Parity also requires the `gcc`, `g++`, `libssl-dev`/`openssl`, `libudev-dev`, and `pkg-config` packages to be installed.

### OS X

On OS X, run this command:

```
curl https://sh.rustup.rs -sSf | sh
```

Parity also requires clang. Clang comes with Xcode command-line tools or can be installed with Homebrew.

### Windows

Make sure you have Visual Studio 2015 with C++ support installed. Next, download and run the rustup installer from `https://static.rust-lang.org/rustup/dist/x86_64-pc-windows-msvc/rustup-init.exe`, start "VS2015 x64 Native Tools Command Prompt", and use the following command to install and set up the `msvc` toolchain:

```
rustup default stable-x86_64-pc-windows-msvc
```

## Downloading, installing and running parity

Now, once you have rust installed on your operating system, you can run this simple one-line command to install parity:

```
cargo install --git https://github.com/paritytech/parity.git parity
```

To check whether parity is installed or not, run this command:

```
parity --help
```

If parity is installed successfully, then you will see a list of sub-commands and options.

# Creating a private network

Now it's time to set up our consortium blockchain. We will create two validating nodes connected to each other using Aura for consensus. We will set up both on the same computer.

## Creating accounts

First, open two shell windows. The first one is for the first validator and the second one is for the second validator. The first node will contain two accounts and the second node will contain one account. The second account of first node will be assigned to some initial ether so that the network will have some ether.

In the first shell, run this command twice:

```
parity account new -d ./validator0
```

Both the times it will ask you to enter a password. For now just put the same password for both accounts.

In the second shell, run this command once only:

```
parity account new  -d ./validator1
```

Just as before, enter the password.

## Creating a specification file

Nodes of every network share a common specification file. This file tells the node about the genesis block, who the validators are, and so on. We will create a smart contract, which will contain the validators list. There are two types of validator contracts: non-reporting contract and reporting contract. We have to provide only one.

The difference is that non-reporting contract only returns a list of validators, whereas reporting contract can take action for benign (benign misbehaviour may be simply not receiving a block from a designated validator) and malicious misbehavior (malicious misbehaviour would be releasing two different blocks for the same step).

The non-reporting contract should have at least this interface:

```
{"constant":true,"inputs":[],"name":"getValidators","outputs":[{"name":"","
type":"address[]"}],"payable":false,"type":"function"}
```

The `getValidators` function will be called on every block to determine the current list. The switching rules are then determined by the contract implementing that method.

A reporting contract should have at least this interface:

```
[
{"constant":true,"inputs":[],"name":"getValidators","outputs":[{"name":"","
type":"address[]"}],"payable":false,"type":"function"},
{"constant":false,"inputs":[{"name":"validator","type":"address"}],"name":"
reportMalicious","outputs":[],"payable":false,"type":"function"},
{"constant":false,"inputs":[{"name":"validator","type":"address"}],"name":"
reportBenign","outputs":[],"payable":false,"type":"function"}
]
```

When there is benign or malicious behavior, the consensus engine calls the `reportBenign` and `reportMalicious` functions respectively.

Let's create a reporting contract. Here is a basic example:

```
contract ReportingContract {
   address[] public validators =
[0x831647ec69be4ca44ea4bd1b9909debfbaaef55c,
0x12a6bda0d5f58538167b2efce5519e316863f9fd];
   mapping(address => uint) indices;
   address public disliked;

   function ReportingContract() {
       for (uint i = 0; i < validators.length; i++) {
           indices[validators[i]] = i;
       }
   }

   // Called on every block to update node validator list.
    function getValidators() constant returns (address[]) {
      return validators;
   }

   // Expand the list of validators.
   function addValidator(address validator) {
      validators.push(validator);
   }

   // Remove a validator from the list.
   function reportMalicious(address validator) {
      validators[indices[validator]] = validators[validators.length-1];
      delete indices[validator];
      delete validators[validators.length-1];
      validators.length--;
   }

   function reportBenign(address validator) {
       disliked = validator;
   }
}
```

This code is self-explanatory. Make sure that in the validators array replaces the addresses with the first address of validator 1 and validator 2 nodes as we will be using those addresses for validation. Now compile the preceding contract using whatever you feel comfortable with.

Now let's create the specification file. Create a file named `spec.json`, and place this code in it:

```
{
    "name": "ethereum",
    "engine": {
        "authorityRound": {
            "params": {
               "gasLimitBoundDivisor": "0x400",
               "stepDuration": "5",
               "validators" : {
               "contract": "0x0000000000000000000000000000000000000005"
                }
            }
        }
    },
    "params": {
        "maximumExtraDataSize": "0x20",
        "minGasLimit": "0x1388",
        "networkID" : "0x2323"
    },
    "genesis": {
      "seal": {
       "authorityRound": {
         "step": "0x0",
          "signature": "0x00000000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000
                00000000000000000000000000000000000"
            }
        },
        "difficulty": "0x20000",
        "gasLimit": "0x5B8D80"
    },
    "accounts": {
        "0x0000000000000000000000000000000000000001": { "balance": "1",
"builtin": { "name": "ecrecover", "pricing": { "linear": { "base": 3000,
"word": 0 } } } },
        "0x0000000000000000000000000000000000000002": { "balance": "1",
"builtin": { "name": "sha256", "pricing": { "linear": { "base": 60, "word":
12 } } } },
        "0x0000000000000000000000000000000000000003": { "balance": "1",
"builtin": { "name": "ripemd160", "pricing": { "linear": { "base": 600,
"word": 120 } } } },
        "0x0000000000000000000000000000000000000004": { "balance": "1",
"builtin": { "name": "identity", "pricing": { "linear": { "base": 15,
"word": 3 } } } },
        "0x0000000000000000000000000000000000000005": { "balance": "1",
"constructor" : "0x606060405260406040519081016040528073831647" },
        "0x004ec07d2329997267Ec62b4166639513386F32E": { "balance":
"10000000000000000000000" }
    }
}
```

Here is how the preceding file works:

- The `engine` property is used to set the consensus protocol and the protocol-specific parameters. Here, the engine is `authorityRound`, which is aura. `gasLimitBoundDivisor` determines gas limit adjustment and has the usual `ethereum` value. In the `validators` property, we have a `contract` property, which is the address of the reporting contract. `stepDuration` is the block time in seconds.

- In the `params` property, only the network ID is what matters; others are standard for all chains.
- `genesis` has some standard values for the `authorityRound` consensus.
- `accounts` is used to list the initial accounts and contracts that exist in the network. The first four are standard Ethereum built-in contracts; these should be included to use the Solidity contract writing language. The fifth one is the reporting contract. Make sure you replace the byte code with your byte code in the `constructor` param. The last account is the second account we generated in the validator 1 shell. It is used to supply ether to the network. Replace this address with yours.

Before we proceed further, create another file called as `node.pwds`. In that file, place the password of the accounts you created. This file will be used by the validators to unlock the accounts to sign the blocks.

## Launching nodes

Now we have all basic requirements ready to launch our validating nodes. In the first shell, run this command to launch the first validating node:

```
parity  --chain spec.json -d ./validator0 --force-sealing --engine-signer
"0x831647ec69be4ca44ea4bd1b9909debfbaaef55c" --port 30300 --jsonrpc-port
8540 --ui-port 8180 --dapps-port 8080 --ws-port 8546 --jsonrpc-apis
web3,eth,net,personal,parity,parity_set,traces,rpc,parity_accounts --
password "node.pwds"
```

Here is how the preceding command works:

- `--chain` is used to specify the path of the specification file.
- `-d` is used to specify the data directory.
- `--force-sealing` ensures that blocks are produced even if there are no transactions.
- `--engine-signer` is used to specify the address using which the node will sign blocks, that is, the address of the validator. If malicious authorities are possible, then `--force-sealing` is advised; this will ensure that the correct chain is the longest. Make sure you change the address to the one you generated, that is, the first address generated on this shell.
- `--password` is used to specify the password file.

In the second shell, run this command to launch second validating node:

```
parity  --chain spec.json -d ./validator1 --force-sealing --engine-signer
"0x12a6bda0d5f58538167b2efce5519e316863f9fd" --port 30301 --jsonrpc-port
8541 --ui-port 8181 --dapps-port 8081 --ws-port 8547 --jsonrpc-apis
web3,eth,net,personal,parity,parity_set,traces,rpc,parity_accounts --
password "/Users/narayanprusty/Desktop/node.pwds"
```

Here, make sure you change the address to the one you generated that is, the address generated on this shell.

## Connecting nodes

Now finally, we need to connect both the nodes. Open a new shell window and run this command to find the URL to connect to the second node:

```
curl --data '{"jsonrpc":"2.0","method":"parity_enode","params":[],"id":0}'
-H "Content-Type: application/json" -X POST localhost:8541
```

You will get this sort of output:

```
{"jsonrpc":"2.0","result":"enode://7bac3c8cf914903904a408ecd71635966331990c
5c9f7c7a291b531d5912ac3b52e8b174994b93cab1bf14118c2f24a16f75c49e83b93e0864e
b099996ec1af9@[::0.0.1.0]:30301","id":0}
```

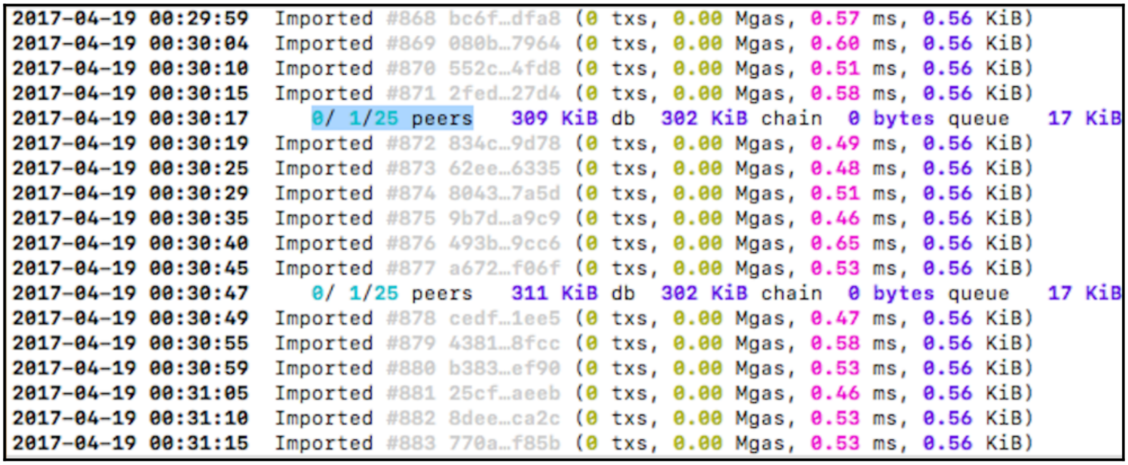
Now run this command by replacing the encode URL and IP address in the `enode` URL to 127.0.0.1:

```
curl --data
'{"jsonrpc":"2.0","method":"parity_addReservedPeer","params":["enode://7ba.
.."],"id":0}' -H "Content-Type: application/json" -X POST localhost:8540
```

You should get this output:

```
{"jsonrpc":"2.0","result":true,"id":0}
```

The nodes should indicate 0/1/25 peers in the console, which means they are connected to each other. Here is a reference image:



```
2017-04-19 00:29:59  Imported #868 bc6f…dfa8 (0 txs, 0.00 Mgas, 0.57 ms, 0.56 KiB)
2017-04-19 00:30:04  Imported #869 080b…7964 (0 txs, 0.00 Mgas, 0.60 ms, 0.56 KiB)
2017-04-19 00:30:10  Imported #870 552c…4fd8 (0 txs, 0.00 Mgas, 0.51 ms, 0.56 KiB)
2017-04-19 00:30:15  Imported #871 2fed…27d4 (0 txs, 0.00 Mgas, 0.58 ms, 0.56 KiB)
2017-04-19 00:30:17     0/ 1/25 peers    309 KiB db  302 KiB chain  0 bytes queue   17 KiB
2017-04-19 00:30:19  Imported #872 834c…9d78 (0 txs, 0.00 Mgas, 0.49 ms, 0.56 KiB)
2017-04-19 00:30:25  Imported #873 62ee…6335 (0 txs, 0.00 Mgas, 0.48 ms, 0.56 KiB)
2017-04-19 00:30:29  Imported #874 8043…7a5d (0 txs, 0.00 Mgas, 0.51 ms, 0.56 KiB)
2017-04-19 00:30:35  Imported #875 9b7d…a9c9 (0 txs, 0.00 Mgas, 0.46 ms, 0.56 KiB)
2017-04-19 00:30:40  Imported #876 493b…9cc6 (0 txs, 0.00 Mgas, 0.65 ms, 0.56 KiB)
2017-04-19 00:30:45  Imported #877 a672…f06f (0 txs, 0.00 Mgas, 0.53 ms, 0.56 KiB)
2017-04-19 00:30:47     0/ 1/25 peers    311 KiB db  302 KiB chain  0 bytes queue   17 KiB
2017-04-19 00:30:49  Imported #878 cedf…1ee5 (0 txs, 0.00 Mgas, 0.47 ms, 0.56 KiB)
2017-04-19 00:30:55  Imported #879 4381…8fcc (0 txs, 0.00 Mgas, 0.58 ms, 0.56 KiB)
2017-04-19 00:30:59  Imported #880 b383…ef90 (0 txs, 0.00 Mgas, 0.53 ms, 0.56 KiB)
2017-04-19 00:31:05  Imported #881 25cf…aeeb (0 txs, 0.00 Mgas, 0.46 ms, 0.56 KiB)
2017-04-19 00:31:10  Imported #882 8dee…ca2c (0 txs, 0.00 Mgas, 0.53 ms, 0.56 KiB)
2017-04-19 00:31:15  Imported #883 770a…f85b (0 txs, 0.00 Mgas, 0.53 ms, 0.56 KiB)
```

# Permissioning and privacy

We saw how parity solves the issues of speed and security. Parity currently doesn't provide anything specific to permissioning and privacy. Let's see how to achieve this in parity:

1. **Permissioning**: A parity network can implement permissioning to decide who can join and who cannot by configuring each node's server to allow connections from only specific IP addresses. Even if IP addresses aren't blocked, to connect to a node in the network, a new node will needs an enode address, which we saw earlier, and that's not guessable. So by default, there is a basic protection. But there is nothing to enforce this. Every node in the network has to take care about this at its end. Similar permissioning for who can create blocks and who cannot can be done through a smart contract. Finally what kind of transactions a node can send is not configurable at the moment.

2. **Identity privacy**: There is a technique to achieve identity privacy by still enabling ownership checks. At the time of setting ownership, the owner needs to specify a public key of an un-deterministic asymmetric cryptography. Whenever it wants ownership checks to pass, it will provide an encrypted form of common text, which will be decrypted by the contract and see whether the account is owner or not. The contract should make sure the same encrypted data is not checked twice.
3. **Data privacy**: If you are using blockchain to just store data, you can use symmetric encryption to encrypt data and store and share the key with people who you want to see the data. But operations on encrypted data is not possible. And if you need operations on input data and still gain privacy, then the parties have to set up a different blockchain network completely.

# Summary

Overall in this chapter, we learned how to use parity and how aura works and some techniques to achieve permissioning and privacy in parity. Now you must be confident enough to at least build a proof-of-concept for a consortium using blockchain. Now you can go ahead and explore other solutions, such as Hyperledger 1.0 and quorum for building consortium blockchains. Currently, Ethereum is officially working on making more suitable for consortiums; therefore, keep a close eye on various blockchain information sources to learn about anything new that comes in the market.

# Index