A Brief Introduction to Web3

Decentralized Web Fundamentals for App Development

—

Shashank Mohan Jain

Apress®

A Brief Introduction to Web3

Decentralized Web Fundamentals for App Development

Shashank Mohan Jain

Apress®

A Brief Introduction to Web3: Decentralized Web Fundamentals for App Development

Shashank Mohan Jain
Bangalore, India

ISBN-13 (pbk): 978-1-4842-8974-7 ISBN-13 (electronic): 978-1-4842-8975-4
https://doi.org/10.1007/978-1-4842-8975-4

Copyright © 2023 by Shashank Mohan Jain

This work is subject to copyright. All rights are reserved by the publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: James Robinson-Prior
Development Editor: James Markham
Coordinating Editor: Gryffin Winkler
Copy Editor: April Rondeau

Cover designed by eStudioCalamar

Cover image by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, email orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

I humbly dedicate all my work to my parents, without whom this would never, ever be possible. I also dedicate this to my wife, Manisha, and my daughter, Isha, for allowing me to take time out of their time to write this book.

Table of Contents

About the Author ix

About the Technical Reviewer xi

Introduction xiii

Chapter 1: Decentralization and Web3 1

1.1 Web 1.0 1

1.2 Web 2.0 1

1.3 Web 3.0 2

1.3.1 Introduction to Decentralization 3

1.3.2 Different Topologies for Networks 4

1.3.3 Decentralized Systems 5

1.3.4 Web3 Case Study 8

1.4 Summary 9

Chapter 2: Blockchain 11

2.1 Types of Blockchains 11

2.1.1 Public Blockchain 12

2.1.2 Private Blockchain 12

2.1.3 Permissioned Blockchain 12

2.2 What Is a Blockchain? 13

2.3 Blockchain Building Blocks 14

2.3.1 Block 14

2.3.2 Chain 15

2.3.3 Network 15
2.4 Where Is Blockchain Used? 16
2.5 Evolution 17
2.6 Consensus 17
2.6.1 Proof of Work 18
2.6.2 Proof of Stake 20
2.7 Blockchain Architecture 21
2.8 Cryptographic Keys 23
2.9 Blockchain Compared to a Singly Linked List 24
2.10 Ethereum 25
2.11 Summary 26
Chapter 3: Solidity 27
3.1 What Is Solidity? 28
3.2 Ethereum 29
3.2.1 Ethereum Virtual Machine 29
3.3 Smart Contracts 30
3.4 Making Sense of Solidity Syntax 30
3.4.1 Pragma 30
3.4.2 Variables 30
3.4.3 Value Types 34
3.4.4 Address 35
3.4.5 Operators in Solidity 35
3.4.6 Loops 37
3.4.7 Decision Flows 39
3.4.8 Functions in Solidity 41
3.4.9 Abstract Contracts 52
3.4.10 Interface 53

3.4.11 Libraries........55
3.4.12 Events........55
3.4.13 Error Handling in Solidity........56
3.4.14 Solidity and Addresses........56
3.5 Summary........62
Chapter 4: Wallets and Gateways........63
4.1 Types of Wallets........64
4.2 So, What Is a Testnet ?........66
4.3 MetaMask........67
4.3.1 Installation........68
4.4 Web3.js........76
4.4.1 web3-eth........77
4.4.2 web3-shh........77
4.4.3 web3-bzz........77
4.4.4 web3-net........78
4.4.5 web3-utils........78
4.5 Infura Setup........79
4.5.1 Interfacing with Ropsten Network via Infura Gateway........82
4.6 Summary........87
Chapter 5: Introduction to Remix IDE........89
5.1 Remix IDE........89
5.2 Creating Own Token........109
5.3 Summary........126
Chapter 6: Truffle........127
6.1 Truffle Installation........127
6.1.1 Installing Node........127
6.1.2 Install Truffle........128

6.2 Smart Contract Deployment via Truffle 129
6.2.1 Contract Code 130
6.2.2 Compile and Deploy the Contract 137
6.3 Summary......144
Chapter 7: IPFS and NFTs 147
7.1 IPFS 147
7.1.1 IPFS: 30,000-Foot View......149
7.1.2 Installation......150
7.2 ERC-721 152
7.3 Creating an ERC-721 Token and Deploying It to IPFS......153
7.4 Summary......165
Chapter 8: Hardhat 167
8.1 Installation of Hardhat Framework 167
8.2 Workflow for Hardhat......167
8.3 Deployment of the Smart Contract......172
8.4 Summary......179
Index......181

About the Author

Shashank Mohan Jain has been working in the IT industry for around 22 years, mainly in the areas of cloud computing, machine learning, and distributed systems. He has keen interest in virtualization techniques, security, and complex systems. Shashank has multiple software patents to his name in the area of cloud computing, Internet of Things, and machine learning. He is a speaker at multiple reputed cloud conferences. Shashank holds Sun, Microsoft, and Linux kernel certifications.

About the Technical Reviewer

Prasanth Sahoo is a thought leader, an adjunct professor, a technical speaker, and a full-time practitioner in blockchain, DevOps, cloud, and Agile, all while working for PDI Software. He was awarded the Blockchain and Cloud Expert of the Year Award 2019 from TCS Global Community for his knowledge share within academic services to the community. He is passionate about driving digital technology initiatives by handling various community initiatives through coaching, mentoring, and grooming techniques. Prasanth has a patent under his name, and to date he has interacted with over 50,000 professionals, mostly within the technical domain. He is a working group member in the Blockchain Council, CryptoCurrency Certification Consortium, Scrum Alliance, Scrum Organization, and International Institute of Business Analysis.

Introduction

We live in exciting times. We are witnessing a revolution in the space of technology every other day. Web3 happens to be one such revolution. With the advent of web3 and blockchain technologies, the way we write applications is changing. The whole idea of this book is to provide the reader with a basic introduction to web3. The book will enable the reader to understand the nuances of web3 and to create simple applications, like creating one's own tokens or NFTs.

CHAPTER 1

Decentralization and Web3

In this chapter, we are going to examine the evolution of Web 3.0 starting from Web 1.0 and 2.0.

1.1 Web 1.0

In Web 1.0, websites delivered static content (rather than dynamic content) written in hypertext markup language (HTML). The data and content were sent by means of a static file system as opposed to a database, and the web pages had only a minimal amount of interactive information.

The following is a list of the principal technologies that were included in Web 1.0:

- HTML (HyperText Markup Language)
- URL encoded in the HyperText Transfer Protocol, or HTTP (Uniform Resource Locator)

1.2 Web 2.0

Since in Web 1.0 most content was static, it gave rise to Web 2.0.

© Shashank Mohan Jain 2023
S. M. Jain, *A Brief Introduction to Web3*, https://doi.org/10.1007/978-1-4842-8975-4_1

The vast majority of us have only experienced the World Wide Web in its most recent iteration, sometimes referred to as Web 2.0, the interactive read–write web, and the social web. Participating in the creative process of Web 2.0 does not require you to have any prior experience as a developer. A great number of apps are designed in such a way that anyone can make their own version of the program.

You are capable of coming up with ideas and communicating those ideas to people around the globe. You are able to post a video to Web 2.0, where it will be available for millions of users to watch, interact with, and comment on. Web 2.0 applications consist of social media sites, such as YouTube, Facebook, Flickr, Instagram, and Twitter, among others. Think about how popular these sites were back when they first launched, and then compare that to how popular they are now.

Web 2.0 allowed applications to scale, but also gave rise to centralized platforms that took control of all user data. This gave them a chance and opportunity to do data mining, which can be put to both positive and negative uses.

Since users were increasingly losing control to these centralized platforms, it led to the emergence of Web 3.0

1.3 Web 3.0

Web 2.0 design inherently depended on centralized systems. Those systems might have been distributed but were not decentralized. This meant that control of those systems remained with individuals or a group of individuals. This led to huge issues as privacy and data usage concerns started to emerge.

This led to the evolution of Web 3.0, which keeps decentralization as its first requirement. This means applications for Web3 will be deployed in a decentralized way and data storage for its apps will also be decentralized. Therefore, it's important to have an idea of what decentralization is before understanding the Web3 landscape.

1.3.1 Introduction to Decentralization

"Unthinking respect for the authority is the greatest enemy of truth." This famous quote by Albert Einstein goes a long way in making us think about whether there should be systems that are controlled by an authority, thereby asking us to have complete faith and trust in the authority, or there should be systems that are decentralized, with consensus as the root of decision making. Central systems, as their name suggests, can be controlled by the whims of a central authority, whereas a decentralized system will favor more inclusive decision making, thereby reducing the chance of corruption within that system.

In a decentralized setup, there is no owner or master. There are only peers participating in the network. These nodes exchange messages between them to form consensus as and when needed. The protocols governing these networks make sure that the overall system of nodes is in a consistent state. Every node in the network has an up-to-date copy of all the data that was recorded. Decentralized networks can also distribute data to validate particular private information without having to hand over that information to a third party. This validation is made possible by the fact that data does not have to be transferred. The validation of data is accomplished by the utilization of a consensus-based technique, wherein the nodes on the network agree to a certain state of the overall system at a certain point in time.

Each participant node in a decentralized network functions independently of the other nodes in the network. Decentralized nodes interact with one another through the use of common standards, but they keep their independence and are in charge of their own privacy management, rather than adhering to the directives of a centralized authority. This not only helps to maintain the network's safety but also ensures that it is governed in a democratic manner.

Figure 1-1 shows how nodes connect and communicate in centralized, decentralized, and distributed networks.

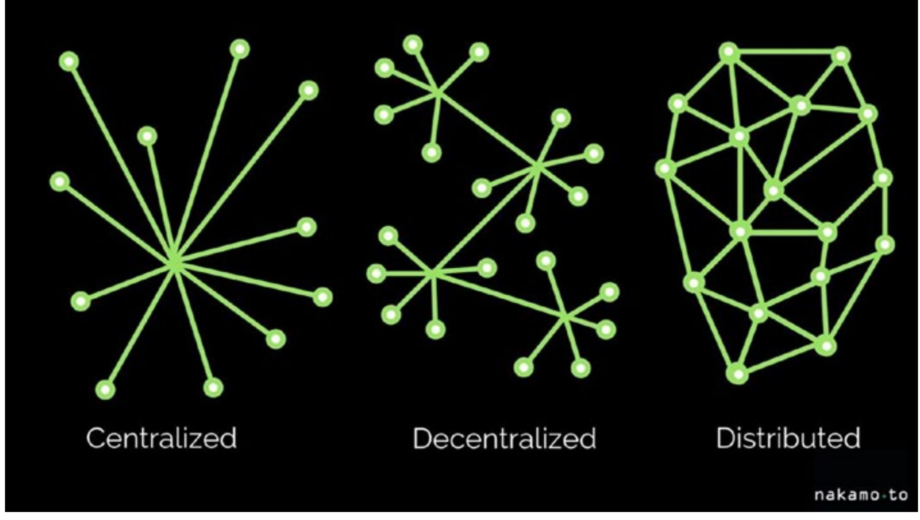

Figure 1-1. *Difference between centralized, decentralized, and distributed networks*

The terms "centralization" and "decentralization" both refer to levels of control. Control is exercised by just one organization or person in a centralized system (a person or an enterprise, for example). There is not a single entity that controls the operation of a decentralized system. Instead, the control is divided among a number of different autonomous entities.

Differences in location are meant to be understood as distribution.

In a system that is not distributed, also known as being co-located, all of the components of the system are situated in the same physical location. A distributed system is one in which different components can be found in different physical locations.

1.3.2 Different Topologies for Networks

Networks can be configured in different topological settings. A few of them are explained in the following sections.

1.3.2.1 Centralized and Non-distributed

In the event that you are putting together a stand-alone application that will run on your Windows computer, this configuration is not distributed and also functions as a centralized system. Your application is under the control of one party (the application vendor), and an additional party, such as Microsoft, is in charge of your operating system (centralized). The application and the operating system are both stored locally on your personal computer, making them both non-distributable components.

1.3.2.2 Centralized but Distributed

Imagine that you are working on a cloud application that is hosted on Amazon Web Services and runs on virtual machines hosted by Amazon Web Services. This configuration is both centralized and distributed at the same time. Your application and operating system are both under the control of AWS (centralized). Cloud storage serves as the host for both the application and the operating system, which may be partitioned across multiple locations.

1.3.3 Decentralized Systems

By its very nature, a decentralized system will be distributed. Peer-to-peer software can be thought of as an example of a decentralized system, which is defined as "a system that requires multiple parties to make their own independent decisions."

It is impossible to have a single, centralized authority in a decentralized system that is responsible for making decisions on behalf of all the different parties. Instead, each party, also known as a peer, is responsible for making locally autonomous decisions in the direction of achieving its own individual goals, which may or may not be in conflict with the goals pursued by other peers. Peers engage in one-on-one communication with

one another, during which they may exchange information or perform a service for other peers. When it comes to decentralized systems, an open system is one in which the participation of peers is not restricted in any way. At any point, a peer may join or leave the system at their discretion.

Before information is added to a blockchain (which is an example of a decentralized system), the nodes participating in the network must use something called a consensus mechanism to reach a decision on whether or not it is accurate. Once the block has been updated with the data, it is sent to all of the nodes in the network. This makes it extremely difficult to alter information that has already been added to the ledger. Mutation of state on a decentralized network requires a majority of nodes to reflect the updated state, which is a very difficult thing to achieve, and therefore the decentralized network of nodes is immutable. This is in contrast to changing the data in a single database that is centralized. If rule of immutability is broken, the validating nodes will disregard such information.

The process of dispersing functions and power away from a central location or authority is known as decentralization. It is difficult, if not impossible, to identify a specific center in a decentralized architecture. The World Wide Web was created as a decentralized platform. Decentralized architectures and systems are demonstrated by blockchain technologies such as Bitcoin and Ethereum.

As the technological landscape changes, decentralized structures emerge. Decentralization has the power to change a lot of things, from governance and industry to justice systems.

Now, with this basic knowledge of decentralization, we can start to understand the Web 3.0 landscape.

Web 3.0 apps are built on blockchains, which are decentralized networks of numerous peer-to-peer nodes. Blockchains are used to record transactions that take place between users. In the context of the Web 3.0 ecosystem, these applications are referred to as decentralized apps (DApps), which is a term that is frequently used. The participants

in the network, known as developers, are paid for their efforts to provide services of the highest possible quality in order to maintain a decentralized network that is both robust and safe.

At a high level, the Web 3.0 architecture comprises four major parts, as follows:

1. Blockchain – A peer-to-peer network of nodes is responsible for the maintenance of these global state machines. Access to the state machine, as well as writing to it, is open to anybody on the globe. In essence, rather than being owned by a single company, it is held collectively by all of the participants in the network. Users are able to upload new data to the blockchain, but they are unable to edit the data that has already been added. There are many types of blockchain available, like Ethereum, Solana, Polkadot, etc.

2. Smart Contracts – The Ethereum blockchain can host something called a smart contract, which is a computer program that runs on the blockchain. These are written by app developers using high-level programming languages such as Solidity or Vyper to specify the logic that lies beneath the state transitions. We will discuss Solidity in Chapter 3.

3. Ethereum Virtual Machine (EVM) – It's a virtual machine that is tasked with putting into action the logic that is outlined in smart contracts. It is in charge of managing the state machine's transitions between states. It generally operates on the bytecode, which is generated by taking the Solidity source code and compiling it via the Solidity compiler.

4. Front End/UI – The user interface (UI) logic is defined by the front end, just like it is in any other application. However, it does communicate with smart contracts, which are programs that describe how an application works.

Before we can develop a simple Web3 app, we need to be conversant with certain technologies and environments. We will start with the Remix integrated development environment (IDE), which we will use as an environment in this chapter for the development and deployment of our Web3 app based on smart contracts.

1.3.4 Web3 Case Study

Every one of us as professionals has heard about LinkedIn, which is a social network for professionals. It is controlled by LinkedIn, and its bosses can decide the rules of the platform. They can also choose whom to censor if they want to. But we don't need to worry. There is a Web3 alternative to LinkedIn that goes by the name of Entre. It can be accessed at https://joinentre.com/.

It's a decentralized professional network built using Web3 technologies. It currently has around 50,000 members and is growing. It provides posting for jobs, calendar for events, and many other features. Since this is a decentralized platform, the rules are hard to change by a central authority.

Apart from this, there are other Web3 platforms, like Deso as an alternate to Twitter and Odysee as an alternate to YouTube. I can say that we are in very exciting times as far as the shape of the technology is concerned.

1.4 Summary

In this chapter, we briefly touched upon the notion of Web 3.0 and decentralization and its forms. In next chapter, we will introduce the reader to the blockchain ecosystem and its various types. We will also study the functioning of some of the existing blockchain networks, like Bitcoin and Ethereum.

CHAPTER 2

Blockchain

Blockchain technology presents an innovative take on the traditional distributed database. The utilization of existing technology in unconventional settings is the source of the innovation. When we think of a traditional database like Postgres, there is a master node that accepts the write requests and is responsible for synchronizing the state changes to other Postgres nodes. In a decentralized setup like blockchain, there is no master node, but still it serves the same purpose of updating the ledger and making sure that other nodes in the network reflect the updated state correctly.

There are a great number of distinct varieties of blockchains and applications for blockchain technology. The blockchain is an all-encompassing technology that is currently being integrated across a variety of different platforms and pieces of hardware around the world.

A blockchain is a data structure that enables the creation of a digital ledger of data and the sharing of that data across a network of parties that are not affiliated with one another. There are a great number of distinct varieties of blockchains.

2.1 Types of Blockchains

Blockchains can be classified under three broader categories.

© Shashank Mohan Jain 2023
S. M. Jain, *A Brief Introduction to Web3*, https://doi.org/10.1007/978-1-4842-8975-4_2

2.1.1 Public Blockchain

Blockchains that are open to the public, like Bitcoin, are large distributed networks that are each managed by their own native token. They have open-source code that is maintained by their community and welcome participation from anyone, regardless of level of involvement.

2.1.2 Private Blockchain

Private blockchains are typically smaller than public ones, and they do not make use of tokens. Their membership is strictly monitored and regulated. Consortiums that comprise reliable members and engage in the confidential exchange of information favor the use of blockchains of this kind. An example of this would be a corporation using a blockchain in its own data center. IBM has an implementation called Hyperledger that is implemented by several corporations.

2.1.3 Permissioned Blockchain

A permissioned blockchain is one in which a user needs permission to access it. This is very different from public and private blockchains. A corporate setup might look for a permissioned blockchain where it uses the blockchain more like a decentralized database within the confines of the corporate boundaries. Examples of permissioned blockchains are Ripple and IBM Food Trust.

The reasons blockchains rose to prominence are as follows:

1. Their decentralized nature – The blockchain setup advocates decentralization in its architecture. This means that in a strong blockchain network like Bitcoin or Ethereum, there are no centralized authorities. Anyone can participate in these

networks by spinning their own node and becoming part of the blockchain network. This is really an empowering feature.

2. Immutable data structure for storage – All data that is stored on the blockchain is permanently recorded across a fleet of nodes/computers across the globe. One can think of this as a permanent ledger of records. Removal of a record from this ledger needs a majority of the network to agree, which is a practical impossibility if the network is big like Bitcoin or Ethereum.

Though the rules vary from blockchain to blockchain, generally any mutation or state chain on the blockchain is done by one entity, which then publishes the changes on the network for other nodes to verify. Only when a majority agrees to that state is the transaction considered valid. The mechanism for who gets the right to change the state can vary from blockchain to blockchain based on the underlying algorithm of the chain. As an example, for a Bitcoin-based blockchain there is the proof-of-work algorithm, which lets the nodes participate in solving a mathematical puzzle, and the node that solves it first gets the right to write on the ledger. The writer then broadcasts the changes to the network for the validator to validate the changes.

2.2 What Is a Blockchain?

A blockchain is a peer-to-peer network that does not rely on a centralized authority to manage the state of the system or ledger. The computers that form the network can be distributed physically. “Full nodes” is another common name for these kinds of computers.

After data has been entered into a blockchain database, it is extremely difficult, if not impossible, to delete or alter that data. This means that once the data is recorded and replicated across tons of nodes across the world, the consumer of the data can have a higher level of trust in the data as compared to data recorded on a centralized ledger. This means that records like property rights, marriage certificates, invoices, and so on can be recorded on the chain for permanence without the worry of tampering. Processes that rely on having a central system in business and banking, such as fund settlements and money wires, can now be completed without having a central authority in place. The implications of having secure digital records are extremely significant for the economy of the entire world.

2.3 Blockchain Building Blocks

There are three main components of a blockchain ecosystem.

2.3.1 Block

A block is a listing of the transactions that have been entered into a ledger over the course of a specific time period. Each blockchain has its own unique dimensions, time intervals, and events that cause blocks to be created.

There are some blockchains whose primary focus is not on maintaining and securing a record of the transactions involving their respective cryptocurrencies. However, every blockchain will record any transactions involving its associated cryptocurrency or token. Consider the transaction to be nothing more than the simple recording of data. The interpretation of what that data means is accomplished by first giving it a value, as is done in the course of a financial transaction.

2.3.2 Chain

A chain can be thought of as a linked list of blocks. The block that comes after the first block (also known as genesis block) will maintain a reference to the previous block and so on and so forth. This reference is maintained by generating a hash of the contents of the previous block. This has some great implications for the immutability of the chain. Anyone who has to alter a previous block will then need to modify all blocks ahead of that particular block. Generally this is a practical impossibility as it might require either huge computing resources in the case of algorithms like proof of work, or a huge stake to be put in the network for proof of stake–based algorithms like Ethereum.

2.3.3 Network

"Full nodes" are the building blocks of the network. Imagine them as a computer that is actively implementing an algorithm to ensure the safety of the network. All nodes in the network have the same copy of records (called the ledger). The ledger comprises all transactions that have ever been recorded on the blockchain.

The network nodes are decentralized and can be run by anyone in any part of the world. Since running these nodes will incur costs, people need some incentive to do so. They are motivated to run a node because they want to earn cryptocurrency, which serves as their incentive. The underlying algorithm of the blockchain provides them with compensation for their service. Typically, a token or cryptocurrency, such as Bitcoin or ether, is awarded as the reward.

2.4 Where Is Blockchain Used?

Applications based on the blockchain are designed around the principle that the network should act as the arbitrator. This kind of system creates an unforgiving and oblivious atmosphere for its users. The computer code effectively becomes the law, and the rules are carried out in the manner in which they were written and are interpreted by the network. The social behaviors and prejudices that are characteristic of humans are absent in computers.

The network is incapable of interpreting the user's intent (at least not yet). As a potential use case for this concept, insurance contracts that are arbitrated on a blockchain have received a lot of attention and research. The ability to keep immutable records is yet another fascinating application of blockchain technology.

You can use them to make a comprehensible timeline of who did what and when. Countless hours are spent by numerous industries and regulatory bodies trying to get a handle on the magnitude of this issue. An immutable ledger will allow us to keep a consistent timeline of transactions that have happened right from the inception of the chain.

Although people frequently use the terms "Bitcoin" and "blockchain" interchangeably, these two concepts are not the same. A blockchain is used by Bitcoin. The underlying protocol that allows for the safe transfer of Bitcoin is known as the blockchain. Bitcoin is the name of the digital currency that serves as the driving force behind the Bitcoin network. So Bitcoin is one of the applications built on top of the blockchain infrastructure.

2.5 Evolution

Blockchain technology was initially developed in conjunction with the cryptocurrency Bitcoin. The whole idea of Bitcoin was to develop a system for decentralized money and banking. The initial purpose of the Bitcoin network's construction was to ensure the cryptocurrency's safety. It has somewhere in the neighborhood of 5,000 full nodes and is spread out across the world. Its primary purpose is to buy and sell Bitcoin and other cryptocurrencies, but the community quickly realized that it could be used for a great deal more than that. Because of its size and the fact that its security has been proven over time, it is also being used to secure other blockchains and blockchain applications that are smaller in scale.

The blockchain technology has undergone a second iteration with the creation of the Ethereum network. It modifies the conventional structure of a blockchain by incorporating a programming language that is embedded within the structure itself. Similar to Bitcoin, it has more than 5,000 full nodes and is distributed across the world. Ether trading, the development of smart contracts, and the formation of decentralized autonomous organizations (DAOs) are the primary uses of Ethereum. Additionally, it is utilized in the process of securing blockchain applications as well as more compact blockchains.

The Ethereum ecosystem is what turned the blockchain ecosystem into a programmable blockchain.

2.6 Consensus

Consensus is one of the most difficult problems to solve in a distributed systems setup.

Blockchains are useful tools because they are capable of self-correcting and do not require the intervention of a third party in order to do so. Through the use of their consensus algorithm, they are able to successfully enforce the rules.

In the world of blockchain technology, “reaching consensus” refers to the procedure of reaching an agreement among a group of nodes that typically do not trust one another. These are the nodes on the network that have full functionality. Transactions that are entered into the network in order for them to be recorded on the ledger are being checked for validity by the full nodes. Each blockchain has its own set of algorithms that it uses to get its network to reach a consensus on the new entries that are being added. Because different kinds of entries are being generated by each blockchain, there is a diverse range of models available for achieving consensus.

Some of the mechanisms used by different blockchains for consensus are as follows:

- Proof of Work – Used by Bitcoin blockchain
- Proof of Stake – Used by Ethereum
- Proof of History – Used by Solana

2.6.1 Proof of Work

The first decentralized cryptocurrency to use a consensus mechanism was Bitcoin, and that mechanism was proof of work. Mining and proof of work are concepts that are closely related to one another. The term “proof of work” comes from the fact that the network needs an extremely high level of processing power to function properly. Proof-of-work blockchains are those in which virtual miners from all over the world compete with one another to see who can solve a mathematical puzzle first. These blockchains are then secured and verified. The victor receives a predetermined amount of cryptocurrency from the network as well as the opportunity to update the blockchain with the most recent transactions that have been verified.

Proof of work offers a number of significant benefits, particularly for the case of Bitcoin, which is a relatively straightforward but extremely valuable cryptocurrency. It is a tried-and-trusted method that can reliably keep a decentralized blockchain in a secure state. As the price of a cryptocurrency continues to rise, more miners will be encouraged to participate in the network, which will result in an increase in both the network's power and its level of security. The Bitcoin network also in its algorithm has a difficulty adjustment mechanism, which is like a feedback loop to alter the difficulty of mining based on demand. If there are huge number of miners and they are able to mine quickly because they have good resources at hand, the algorithm increases the difficulty level of the mining to keep the mining time of one block to 10 minutes.

The mathematical puzzle used by a proof-of-work algorithm is not truly a puzzle in the sense of a physical puzzle. It uses the one-way function of cryptographic hashes as the underlying mechanism. Cryptographic hashes are designed to be deterministic one-way functions. What this means is that any content, be it text of few lines or document or images or videos can be reduced to a specific size. For instance, in a SHA 256–based hash function, the size of the output is 256 bits, no matter how big or small the input is. The other property of hash functions is that the same content will generate the same hash no matter how many times you run the same hash function over it. This is where the determinism comes from.

As I said, it's a one-way function, so going from input to output is easy. Going back from output to input is impossible, or one could say computationally infeasible. Bitcoin uses this property for its proof-of-work algorithm.

So basically the Bitcoin network provides a hash value as the input and asks the miners to generate a hash by taking the following two inputs:

- The block header
- A random number called nonce

The miner needs to add these two together and generate a hash that is less than the value provided by the Bitcoin network for a specific block. As we discussed, since hashing is a one-way function, the miner now starts to add different values of nonce to the header and has to effectively generate the hashes to make it less than the target hash provided by the system. There is no other way to do this except by crunching these hashes.

2.6.2 Proof of Stake

Ethereum blockchain started with proof of work as the algorithm but now is moving to a proof-of-stake algorithm. In proof of stake, to mine the ether (that's the currency on the Ethereum blockchain) the network participant has to stake the currency. Say, for example, I want to be a validator on the network. I then have to put some money at stake to validate the transactions. This puts my skin in the game, and I am disincentivized to cheat. So, instead of putting in energy, as in the case of Bitcoin, Ethereum advocates putting money itself at stake for being a participant on the network.

From the pool of stakers, the one who puts the maximum stake will be chosen as the validator of transactions and will be rewarded for the task.

Other validators will be able to do a proof of validity of the block of transactions once the winner has validated the most recent block of transactions. The blockchain is updated whenever the network reaches a certain predetermined number of confirmations or attestations.

One of the most significant distinctions between the two consensus mechanisms is the amount of energy required. Proof-of-stake blockchains enable networks to function with significantly lower resource consumption than blockchains based on proof of work. The debate still rages as to which is a better mechanism as far as representation of money is concerned.

Both of these consensus mechanisms have economic repercussions that punish malicious actors for disrupting the network and discourage others from doing so. The sunk cost of computing power, energy, and

time is the punishment for miners who submit invalid information or blocks in proof-of-work systems. This is the case for Bitcoin and other cryptocurrencies.

The main idea is to create a heavy penalty for cheating the system. In the case of proof of work, it's the energy spent that is at stake, whereas in proof of stake it is the money that is put at stake. If, say, a network participant is found to have accepted a corrupt block, a portion of the funds that they have staked will be "slashed" as a form of punishment. The network will determine the maximum amount by which a validator's reward can be reduced.

2.7 Blockchain Architecture

A client–server network is used in the conventional design of the World Wide Web's architecture. Due to the fact that it is a centralized database that is controlled by a number of administrators who each have permission to make changes, the server in this scenario stores all of the necessary information in a single location so that it can be easily updated.

When it comes to the distributed network architecture of blockchain technology, every participant in the network is responsible for the maintenance, approval, and updating of new entries. Not only are there a number of different people who have control over the system, but everyone who is part of the blockchain network does. Every member is responsible for ensuring that all of the records and procedures are in order, which ultimately leads to the validity and safety of the data. Therefore, it is possible for parties that do not necessarily trust one another to come to an agreement with one another.

In a nutshell, the blockchain can be described as a decentralized distributed ledger of various transactions that is organized into a P2P network. This ledger can be made public or kept private. This network is

made up of a large number of computers, but it is constructed in such a way that the data cannot be changed unless there is agreement from all of the computers in the network (each separate computer).

A list of blocks containing transactions arranged in a specific sequence is what the blockchain technology uses to represent its underlying structure. These lists may be kept in the form of a straightforward database or as a simple text file (using the txt format). One can think of the blockchain as a persisted linked list of transaction blocks.

We have already covered the concepts of nodes, blocks, and chains in the context of blockchains. There is one more important entity that needs a mention. This entity is the miner. As an example of the Bitcoin blockchain, the miner is the entity that is responsible for mutating the state of the blockchain. This mutation is achieved by adding a new block to the chain. To get permission to mutate the state, the miner has to solve a mathematical puzzle. The puzzle is trying to find the number for a system-provided hash. So the miner has to run through all different combinations of numbers to see if the hash for one of them matches. This computation is run by many miners across the globe and whoever wins the race gets the permission to write the new block. Once the miner writes the block they publish this on the network for the verifiers to verify the block. The verification process is simple as the miner will provide the hash as well as the number for which this hash was generated. So verification is to just calculate the hash for the number and compare it with the hash the system has asked for. Once a majority of miners has verified the block, this block gets replicated across the network and gets added to all computers on the network gradually.

At a high level, the transaction flow in a Bitcoin blockchain works like the one shown in Figure 2-1.

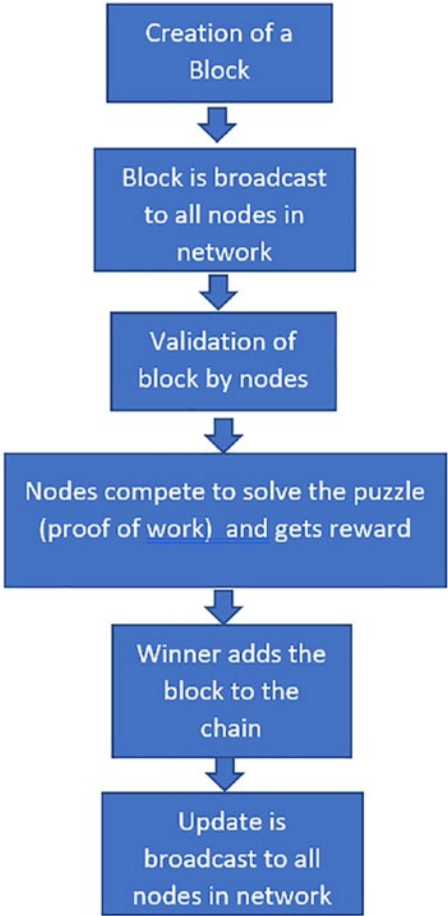

Figure 2-1. *How a transaction works on Bitcoin blockchain*

2.8 Cryptographic Keys

Since public blockchain networks are permissionless and trustless, we need a mechanism to authenticate the users in absence of a centralized authority. This task is achieved by providing each participant on the network with a private and public key pair. The private key remains with

the user and should be kept highly protected. The private key is used to sign the transactions of the user, whereas the public key (which is visible to everyone) can be used to validate the signer. In this way, without a central authority and using the private and public key infrastructure, security is achieved on the decentralized blockchain network.

2.9 Blockchain Compared to a Singly Linked List

There is a certain degree of similarity between the data structure of a blockchain and that of a singly linked list, as shown in Figure 2-2.

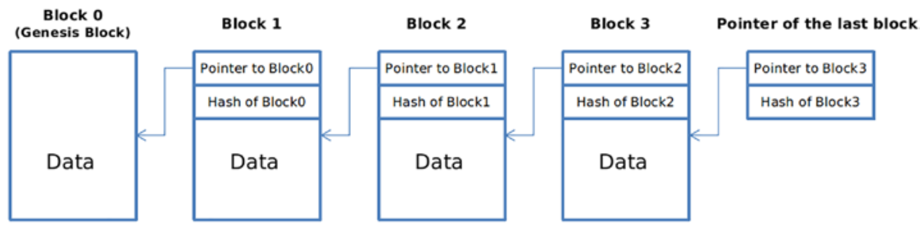

Figure 2-2. *Linked list–like structure depicted for the blockchain*

We can see the first block is called the genesis block, and then from there each new block created holds a reference to the previous block. As an example, block 1 will hold a reference to block 0 and block 2 to block 1 and so on. Apart from this pointer, the hash of the previous block is stored. This appears like a singly linked list data structure. To add to the list we need to use either proof of work or proof of stake, but deletion is extremely difficult on the blockchain, unlike a linked list, where it's easy to remove nodes.

In essence, a blockchain is immutable. No record can be deleted. If we try to think about it, any alteration to any block means all the blocks ahead of that block have to be updated with the hash of previous blocks.

In the case of networks like Bitcoin, this would mean needing immense computing power to use proof of work to alter the blocks. That's one of the reasons it's almost impossible to attack the Bitcoin blockchain network.

Next, we are going to introduce the Ethereum blockchain, which is the programmable blockchain we will use for all our future work in this book.

2.10 Ethereum

Ethereum is a blockchain network that includes a Turing-complete programming language that can be used to build a variety of decentralized applications (also called DApps). The Ethereum network is powered by its own cryptocurrency, ether. The Ethereum network is currently well known for enabling the use of smart contracts. Smart contracts can be compared to cryptographic bank lockers that contain specific values. Certain conditions must be met before these cryptographic lockers can be unlocked. Solidity, a programming language, is primarily used to create smart contracts. Solidity is a relatively simple-to-learn object-oriented programming language. We will see more of Solidity in Chapter 3.

Ethereum works on two types of accounts:

1. **Externally owned accounts (EOA)**

   Private keys are used to control externally owned accounts. Each EOA is protected by a public–private key pair. Users can communicate by creating and signing transactions.

2. **Contract accounts**

   Contract codes are used to manage contract accounts. These codes are saved alongside the account. Each contract account has an associated ether balance. These accounts' contract codes are activated whenever they receive a transaction from an EOA or a message from another contract. When the contract code is enabled, it is possible to read/write messages to local storage, send messages, and create contracts.

2.11 Summary

In this chapter, we covered the basics of blockchain architecture and the different consensus algorithms like proof of work and proof of stake that are commonly used in blockchain networks like Bitcoin and Ethereum. In the next chapter, we will cover Solidity, which is the language used on the Ethereum blockchain, and how Solidity is used for creating smart contracts.

CHAPTER 3

Solidity

Have you ever heard of or encountered a smart contract? If someone has been living under a rock, they maybe have not heard about it. Having said that, smart contracts define a way by which we can execute code on the blockchain.

One of the languages used to define these smart contracts is Solidity. Solidity allows us to program the Ethereum blockchain, thereby opening up doors for the immense possibilities of decentralized application development. It's important that blockchain beginners learn Solidity. It is essential to have a thorough understanding of how Solidity may be utilized for the development of smart contracts, as well as to take an in-depth look at the various components.

Learners should also reflect on examples in order to gain a better understanding of the components that make up the architecture and operation of Solidity. In the same vein, an in-depth contemplation of the various applications of Solidity could be of use in gaining a better understanding of its significance. In addition, a lot of students wonder, "Is it easy to study Solidity?" It is essential to be aware that it is simpler to learn about Solidity if you have a good tutorial to follow along with.

This chapter provides a comprehensive summary of Solidity as well as other related elements, such types, functions, events, and inheritance.

© Shashank Mohan Jain 2023
S. M. Jain, *A Brief Introduction to Web3*, https://doi.org/10.1007/978-1-4842-8975-4_3

3.1 What Is Solidity?

Solidity is a high-level programming language whose primary purpose is to facilitate the creation and execution of smart contracts. C++, JavaScript, and Python are the key programming languages that have had an impact on Solidity. Solidity was proposed by Gavin Wood and was written by the Ethereum team under Christian Reitwiessner. Additionally, Solidity was designed with a focus on the Ethereum virtual machine.

On the Ethereum blockchain, the creation of decentralized applications, often known as DApps, can be accomplished with the help of Solidity. The year 2015 marked a turning point for Solidity. In this condensed version, the primary highlighted qualities of Solidity are essential components that demonstrate the efficacy of Solidity in various contexts. When you study Solidity, you will notice that it has the following significant qualities:

1. Smart contracts are implemented using Solidity, a statically typed language. Smart contracts can be created and deployed using an object-oriented or contract-oriented framework.
2. It is possible to create contracts that deal with voting, consensus, multisignature wallets, and other applications using the Solidity programming language.

Before we start to understand the nuances of Solidity, we need to be aware of a few things.

3.2 Ethereum

Using Ethereum as a starting point for learning Solidity is a no-brainer. The Ethereum virtual machine is the primary aim of Solidity, which means that readers should pay attention to Ethereum in the context of Solidity. This open-source and decentralized platform helps execute smart contracts and is built on top of blockchain technology. Ethereum is an open software platform that makes use of blockchain technology to assist developers in the creation and deployment of decentralized applications.

Ether is the crypto asset that powers the Ethereum network. Application developers use ether to pay for transaction services and fees on the Ethereum network, making it more than just a transferable cryptocurrency.

Ethereum uses a second token type to pay miners for the inclusion of transactions in specific blocks of the Ethereum blockchain. For all transactions involving smart contracts, gas is an essential component, and Ethereum's gas token serves as a vital link in the chain. Ethereum gas is a key factor in attracting miners that want to plant a smart contract on the blockchain.

3.2.1 Ethereum Virtual Machine

The Ethereum virtual machine (EVM) is a crucial component of the Ethereum blockchain. Smart contracts in Ethereum can be executed in this virtual machine. Without EVM, it is impossible for a global network of public nodes to provide the necessary security and capability for running untrusted code.

Now, since we can run untrusted code, the environment should provide us with the right security measures, like protection against a denial of service attack. Ethereum, by using the concept of gas, protects against such attacks.

3.3 Smart Contracts

All of the business logic that users write in their applications should be defined within smart contracts. As a result, it's critical to get a firm grasp of the fundamentals of smart contracts as soon as possible.

Before writing a smart contract, one should grasp the basics of the Solidity programming language. That is what we will do here.

3.4 Making Sense of Solidity Syntax

3.4.1 Pragma

The pragma directive is the first line of code in a Solidity smart contract; for example, the pragma directive in Solidity 0.4.16 source code samples like this one. That's not all: The contract is compatible with Solidity versions higher than the specified one. Additionally, the smart contract's pragma directive confines Solidity to version 0.9.0, as well.

```
Example of pragma syntax
pragma solidity ^0.8.13;
```

3.4.2 Variables

Any computer language would be incomplete without variables. As a result, you'll need a Solidity course to learn about variables and how they may be used to store information. Remember that variables are nothing more than placeholders in memory for values to be stored. As a result, by declaring a variable, you're reserving memory space in advance.

Solidity supports different data types, such as integer, string , Boolean, etc. Depending on the variable's data type, the operating system determines how much reserved memory should be allocated and which entities should be stored there.

Learning Solidity necessitates an understanding of variables. Variables in Solidity can be divided into three categories: state dependent, locally dependent, and globally dependent. State variables are those that are permanently held in contract storage. During the process of executing a function, the values of the local variables are present.

Global variables, on the other hand, are unique variables found in the global namespace that aid in the retrieval of blockchain-related data. As a statically typed language, Solidity necessitates specifying the type of a state or local variable when it's defined. In the absence of any concept of "null" or "undefined," all declared variables have a default value assigned to them based on their type. Some further thoughts on variables in Solidity are warranted here, as follows:

1. State Variables – Variables that are stored in contract storage.

   As an example:

   ```
   pragma solidity ^0.8.13;
   contract NewsBlog {
      uint newsCount;      // newsCount is a State
                              variable
      constructor() public {
         newsCount = 1;   // Initializing the state
                             variable
      }
   }
   ```

2. Local Variables – Variables whose values are preserved throughout the execution of the function.

   As an example:

   ```
   pragma solidity ^0.8.13;
   ```

```
contract NewsBlog {
   uint newsCount;      // newsCount is a State
                           variable
   constructor() public {
      newsCount = 1;   // Initializing the state
                          variable
   }
function addToNews() public view returns(uint)
{
uint news=1; //news is an example of a local variable
used within a function scope
uint res=newsCount+news;
return res;
}
}
```

3. Global Variables – The global namespace contains special variables for obtaining blockchain-related data.

Here is a list of all of Solidity's global variables and what they do:

1. This is what `blockhash(uintblockNumber)` gives back: Block hash representation; only works for the 256 most recent blocks; current blocks are ignored.

2. Coinbase's payment address is `block.coinbase`. Displays the current block's address.

3. Difficulty of a block is indicated by the `block.difficulty (uint)`.

4. The `uint` value of the function `block.gaslimit`. The current block's gas limit is displayed.

5. The `uint` value of `block.number`. The current block number is displayed here.

6. The current time as a universal identifier (`uint`): The current block's timestamp is displayed in terms of the number of seconds since the unix time.

7. `msg.sig` – You can get the first four bytes of a function's identification or call data by using this method.

8. `msg.data` (bytes of call data)

9. If you want to know the current block timestamp, you can use the `now` function.

3.4.2.1 Variable Naming

The following guidelines should be kept in mind while naming your variables in Solidity:

1. The Solidity reserved keywords should not be used as a variable name. The next part includes some discussion of these crucial terms. Variable names such as `break` or `boolean`, for example, are invalid.

2. Variable names in Solidity should not begin with a numeric (0–9). They can't begin with anything other than a letter or the underscore. A variable name such as `543coin` is invalid, while `_543coin` is acceptable.

3. Capitalization is important when naming Solidity variables. `Blog` and `blog` will be treated as two different variables.

3.4.2.2 Scope of Variables

Local variables are local to the function, which means any effect on their state remains within the confines of that method, whereas state variables can have three different kinds of effects:

- Public – State variables that are public can be accessed both internally and through messages. A getter function is automatically made for a state variable that is available to the public.
- Internal – These state variables can only be accessed within the same contract or from within a child contract of this contract.
- Private – These are very confined in scope—only to the same contract—and cannot be accessed even from a child contract.

3.4.3 Value Types

You can learn about the various data types in the Solidity blockchain programming language by taking a thorough look at any tutorial on the subject. Solidity supports a wide range of predefined and custom data types. The following are some of the most common data types found in Solidity:

- Integers, both signed and unsigned
- Boolean
- Fixed-point numbers, both signed and unsigned, of various sizes

- Fixed *X*x*Y* fixed-point number, where *X* defines the number of bits accounted for by type and *Y* accounts for the number of decimal places.
- Unfixed *X*x*Y*, where *Y* represents an unsigned fixed-point number.

3.4.4 Address

An Ethereum address is represented by a 20-byte value called an "address" in the Solidity tutorial. To get the balance, you can use the method `.balance` with an address. The `.transfer` function could be used to transfer the balance to another address.

3.4.5 Operators in Solidity

As with other programming languages, Solidity has support for operators. We will illustrate a few such operators here.

3.4.5.1 Arithmetic Operator

Arithmetic operations comprise the following:

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Increment (++)
6. Decrement (--)
7. Modulus (%)

3.4.5.2 Comparison Operator

Comparison operators comprise the following:

1. Equal (==) – To check if two operands are equal or not
2. Not Equal (!=) – To check if two operands are not equal
3. Greater than (>) – To check if operand on left of operator is greater than one on right
4. Greater than or equal to (>=) – To check if operand on left of operator is greater than or equal to one on right
5. Less than (<) – To check if operand on left of operator is less than one on right
6. Less than or equal to (<=) – To check if operand on left of operator is less than or equal to one on right

3.4.5.3 Logical Operators

Logical operators in Solidity comprise the following:

1. Logical And (&&) – If operands on left-hand side of operator are positive, this returns true.
2. Logical Or (||) – If one of the operands is positive, this returns true.
3. Logical Not (!) – It reverses the logical state of operand and returns the reverse of the operand state.

3.4.5.4 Assignment Operators

Assignment operators in Solidity comprise the following:

1. Simple Assignment (a=b+c)
2. Add and Assign (a+=b is equivalent to a=a+b)
3. Subtract and Assign (a-=b implies a=a-b)
4. Multiply and Assign (a*=b implies a=a*b)
5. Divide and Assign (a/=b implies a=a/b)
6. Modulus and Assign (a%=b implies a=a%b)

3.4.6 Loops

Loops are a simple construct that allows a repetition to happen in the code. Say, for example, we want to display a count number of blog hits for our blog web page. We can use a loop to iterate over the hits of the blog web page and display the result.

Solidity, like other programming languages, has support for different syntaxes for the loops.

Here is an example of a `for` loop in Solidity:

```
pragma solidity ^0.8.13;

contract News {
   uint newsCount;
   constructor() public{
      newsCount = 0;
   }
   function count() public
      returns (uint val) {
     //for loop example
     for(uint i=0; i<5; i++){
```

```
   newsCount++;
   }
   return newsCount;
   }
}
```

```
While loop in solidity
pragma solidity ^0.8.13;

contract News {
   uint newsCount;
   constructor() public{
      newsCount = 0;
   }
   function count() public
      returns (uint val) {
      uint i;
      while (i<=5){
        newsCount++;
        i++;
        }
        return newsCount;
   }
}
Do while loop in solidity
pragma solidity ^0.8.13;

contract News {
   uint newsCount;
   constructor() public{
      newsCount = 0;
   }
```

```
   function count() public
      returns (uint val) {
     uint I;
     do {
        newsCount++;
        i++;
        }
        while (i<=5);
        return newsCount;
   }
}
```

3.4.7 Decision Flows

All programming languages, including Solidity, have a means to apply conditional logic in the code. Based on certain conditions, we might choose to take some path in code versus another path based on some other condition's being true. This is what decision flows allow us to accomplish.

Solidity has support for `if` and `if else` statements to accomplish decision flows and conditional logic within smart contracts.

Here is an example of an `if` statement:

```
pragma solidity ^0.8.13;

contract News {
   uint newsCount;
   constructor() public{
      newsCount = 0;
   }

   function isFake() public returns (string memory)
   {
```

```
//here we check the if condition
   if (newsCount>3) return "too many news articles";
   return "not much news";
   }

   function count() public
      returns (uint val) {
     uint i;
     do {
        newsCount++;
        i++;
        }
        while (i<=5);
        return newsCount;
   }
}
```

Similarly, we have the `if else` flow for decision making. For example:

```
pragma solidity ^0.8.13;

contract News {
   uint newsCount;
   constructor() public{
      newsCount = 0;
   }

   function isFake() public returns (string memory)
   {
//if else condition check here
   if (newsCount>3)
{
return "too many news articles";
```

```
}
   else
   {
   return "not much news";
   }
   }

   function count() public
      returns (uint val) {
     uint i;
     do {
        newsCount++;
        i++;
        }
        while (i<=5);
        return newsCount;
   }

}
```

3.4.8 Functions in Solidity

Solidity advocates reusability and readability in code by allowing the creation of functions within smart contracts. Functions allow us to provide readable code and also allow us to have a certain functionality defined as a chunk of code. This means that you don't have to write the same code over and over again. It helps programmers write code that is easy to change. Functions let a programmer break up a big program into smaller pieces that are easier to work with.

Solidity has all the features that are needed to write modular code with functions, just like any other advanced programming language.

Let's look at an example of a `count` function within a smart contract. We are not showing the complete contract here but only the function definition:

```
function count() public
      returns (uint val) {
     uint I;
     do {
        newsCount++;
        i++;
        }
        while (i<=5);
        return newsCount;
   }
```

Here, the function name is `count`, and it doesn't take any parameters. It returns an integer value, and its scope is public.

We can also define functions that take input values as parameters.

Here is an example of a function with parameters:

```
function oddEven(uint x) public returns (string memory)
   {
   if (x%2==0) return “even” ;
   else
   {
   return "odd";
   }
   }
```

Here, we see the function `oddEven` takes a parameter that is an integer and returns a string as to whether the number is odd or even.

A Solidity function can have a `return` statement, but it doesn't have to. If you want a function to return a value, you need to do this. The last statement in a function should be this one.

3.4.8.1 Function Modifiers

A function modifier is an important construct in Solidity. It allows us to make code more reusable by externalizing some of the common aspects of code in a modifier. As an example, if we wanted to print a message before and after a function invocation, one way would be to put this code in each and every function body. Another way would be to create a modifier that acts as a placeholder for a function and allows us to weave some aspects around the code.

Modifiers are most often used to check a condition automatically before a function is run. As an example, if we wanted to authenticate a user before letting the invocation of the function happen, we would put the authentication logic inside of a modifier. Before the function got invoked, this authentication code would be executed. If authentication were to succeed, the function would be executed or else an exception would be thrown.

Here is an example of a modifier:

```
modifier SampleModifier {
    // logic for the modifier
}
```

A modifier can also take input parameters; for example:

```
modifier SampleModifier (string x) {
    // logic for the modifier
}
```

Let's examine the workings of the modifier. Here, we create a modifier by name of `checkOwner`:

```
modifier checkOwner {
```

```
    require(msg.sender == owner);
    _;
}
```

The reader can see there is `_;` in the code of the modifier. This is known as a merge wildcard. One can treat this as a placeholder for a function to be executed. To explain it clearly, this is the place where the function to which this modifier is attached will be executed. This gives us an ability to place some generic code before and after the function's execution. A reader familiar with aspect-oriented programming can draw a parallel here. In order to work, a modifier must have the symbol `_;` in its body.

Here, we show how the `_;` can be placed in a modifier:

```
modifier executeBefore {
    require(/*execute code before the function*/);
    _; // here the _ signifies the actual function to be
    executed.
}
```

This example is where post-processing is done via the modifier. Here, the function's execution happens before the modifier is invoked:

```
modifier executeAfter {
    _; // execute the actual function
    require(/* execute code after the function*/)
}
```

Here is an example of the usage. The function `isValidUser` needs to return `true` only if a user is valid. We have two choices here. Either we place the user validity check in every function that needs this, or we externalize it via the modifier and also have clear separation of concerns. We think using the modifier is a cleaner and better approach. Here, we show how this can be achieved via the modifier:

```
function isValidUser(address _user) public view returns(bool) {
    // logic that checks that _user is valid
    return true;
}
modifier CheckUser {
    require(isValidUser(msg.sender));
    _;
}
```

There is also a way to apply multiple modifiers to a function:

```
contract Sample {

    modifier A() {
        require(
            //some checks
        );
        _;
    }
    modifier B() {
        require(
            //some more checks
        );
        _;
    }
    function test() public B() A() {
        //function code
    }
}
```

The modifiers get executed from left to right, which means modifier A gets executed first and then modifier B.

There are more details related to modifiers that are beyond the scope of this book.

Next, we look at view functions

3.4.8.2 View Functions

In Solidity, a view function is a function that only reads the contract's state variables and doesn't change them.

If any of the following statements are in a view function, the compiler will think that they are changing state variables and give a warning:

- Modify/overwrite state variables.
- Create new contracts.
- Invoke a function that is not pure or view.
- Emit event.
- Use certain opcodes in inline assembly.
- Use self-destruct.
- Use low-level function calls.
- Send ether along with function calls.

Here is an example:

```
pragma solidity ^0.8.13;
contract Sample {
  // state variable
  uint x = 10;

  // define a view function
  // it returns product of two numbers that are passed as
     parameter
  // and the state variable x
```

```
  function calculateProduct(uint y, uint z) public view
  returns(uint) {
    uint product = x*y*z;
    return product;
  }
}
```

As we can see in the code, we use the number *x*, which is a state variable only in read-only mode, and don't modify it.

3.4.8.3 Pure Functions

In Solidity, pure functions don't read or change the state variables. Instead, they only use the parameters passed to the function, or any local variables it has, to figure out what the values are. If there are statements in pure functions that read state variables, access the address or balance, access any global variable like `block` or `msg`, call a function that isn't pure, and so forth, the compiler will issue a warning.

For example:

```
pragma solidity ^0.8.13;

contract Sample {
  // state variable
  uint x = 10;

  // define a pure function
  // it returns product of two numbers that are passed as
  parameter

  function calculateProduct(uint y, uint z) public pure
  returns(uint) {
    uint product = y*z;
    return product;
```

```
    }
}
```

We can see that we only use parameters to calculate the product; the state variable is not even read within the function.

3.4.8.4 Fallback Function

A fallback function in Solidity is an external function that doesn't have a name, any parameters, or any return values. As an example, a fallback function is responsible for sending, receiving, and holding the ethers in the smart contract in an Ethereum blockchain. In one of the following situations, it is carried out:

- If a function identifier doesn't match any of a smart contract's functions
- If the function call did not come with any data

A fallback function has the following properties:

- They are unnamed functions.
- They cannot accept arguments.
- They cannot return anything.
- There can be only one fallback function in a smart contract.
- It is compulsory to mark it `external`.
- It should be marked as payable. If not, the contract will throw an exception if it receives ether without any data.
- It is limited to 2300 gas if invoked by other functions. Don't worry about the gas aspects as of now. We will

cover this in detail when we have an understanding of Ethereum.

For example:

```
pragma solidity ^0.8.13;

contract Sample {
   uint  public z ;
   fallback () external { z = 100; }
}

contract Invoker {
   function callSample(Sample sample) public returns (bool) {
     // here we invoke a function that doesn't exist
      (bool success,) = address(sample).call(abi.encodeWithSign
      ature("hello()"));
      require(success);
      // sample.z is now 100

   }

}
```

Here, we can see the caller contract tries to call a `hello` function on contract Sample. Since the function by name of `hello` doesn't exist in Sample contract, the fallback function is invoked, which sets the value of state variable *z* to 100.

3.4.8.5 Function Overloading in Solidity

Function overloading allows us to have multiple functions with the same name within the same contract. This is only possible if the functions' types or arguments are defined differently.

Here is an example of a function overloading:

```
pragma solidity ^0.8.13;

contract Sample {
   function calculateProduct(uint x, uint y) public pure
   returns(uint){
      return x*y;
   }
   function calculateProduct(uint x, uint y, uint z)
   public pure returns(uint){
      return x*y*z;
   }
   function doProductUsing2Arguments() public pure
   returns(uint){
      return calculateProduct(4,5);
   }
   function doProductUsing3Arguments() public pure
   returns(uint){
      return calculateProduct(4,5,6);
   }
}
```

In all the preceding examples, we saw the usage of contracts. Here, we define what a Solidity contract is.

One can treat a contract in Solidity similar to a C++ class. A contract in Solidity has the following:

1. Constructor
2. State variables
3. Functions

There are different visibility requirements for functions and state variables. These are accomplished by visibility quantifiers, which are shown here:

1. External – Other contracts are supposed to call external functions. They can't be used to call inside the company. To call an outside function from within this contract, a call to the function name is needed.
2. Public – These functions can be invoked both from within the same contract or from other contracts. Solidity automatically makes a getter function for every public state variable.
3. Internal – Contrary to public functions, the visibility of internal functions is scoped to the same contract in which they are defined or to the child contract of the parent within which this internal function is defined.
4. Private – Private functions and variables can only be used within the same contract. Even child contracts can't use them.

Solidity contracts can be extended in functionality by using the concept of inheritance.

A simple example of inheritance is shown here:

```
pragma solidity ^0.8.13;

contract Sample {
   constructor() public {

   }
```

```
   //we define a public function to show inheritance
   function getProduct(uint p, uint q) internal pure returns
   (uint) { return p * q; }
}
//Derived Contract
contract ChildSample is Sample {
   uint private res;
   Sample private samp;
   constructor() public {
      samp = new Sample();
   }
   function getProductOfTwo() public {
    //invoke the parent getProduct function
      res = getProduct(6, 7);
   }

}
```

3.4.9 Abstract Contracts

A contract is abstract if it has at least one function that doesn't have an implementation. This kind of contract is used as a starting point. Most of the time, both implemented and abstract functions are part of an abstract contract. Derived contracts will use the existing functions when needed and implement the abstract function. If a derived contract doesn't implement the abstract function, it will be marked as abstract.

Here is an example:

```
pragma solidity ^0.8.13;
abstract contract  Sample {
   function  getProduct() virtual public view returns(uint);
```

```
}
contract SimpleSample is Sample {
   function getProduct() public override view returns(uint) {
      uint x = 10;
      uint y = 11;
      uint product = x*y;
      return product;
   }
}
```

In this code, we define an abstract contract `Sample` with one function `getProduct`. Note the `getProduct` function is not implemented in the abstract contract `Sample` as desired.

3.4.10 Interface

Interfaces are like abstract contracts, and the `interface` keyword is used to make them. Here are the most important parts of an interface. One can think of an interface as a contract that can be implemented in the way the implementer decides. As an example, the ERC20 interface defines a set of methods that can then be implemented for creating tokens on the Ethereum blockchain. A contract can inherit from one or multiple interfaces. So one can combine an implementation using multiple interfaces. The following are also true:

- Interfaces cannot have implemented functions.
- External is the only type of function that can be part of an interface.
- There can't be a function `Object() { [native code]` } in an interface.
- There can't be any state variables in an interface.

- Using interface name dot notation, you can get to enums and structs that are part of an interface.

For example:

```
pragma solidity ^0.8.13;

interface Sample {
   function getProduct() external view returns(uint);
}
contract SimpleSample is Sample {
   function getProduct() public view returns(uint) {
      uint x = 10;
      uint y = 11;
      uint product = x*y;
      return product;
   }
}
```

Apart from abstract contracts and interfaces, Solidity also supports the concept of libraries

3.4.11 Libraries

Libraries are like contracts, but they are mostly made to be used again. A library has functions that can be called by other contracts. There are some rules about how a library can be used with Solidity. Here are some of the most important things about a Solidity library:

- If a library function does not change the state, it can be called directly.

- Solidity libraries are meant to be stateless, so there can't be any state variables for a library.
- As far as inheritance goes, a library cannot inherit a variable, and the library itself cannot be inherited.

Solidity also provides the concept of events.

3.4.12 Events

An event is a component of a contract that can be inherited. When an event is fired, the arguments that were supplied to it are written down in the transaction log. These logs are kept on the blockchain, and until the contract is added to the blockchain, they can be accessed by using the address of the contract. It is not possible to access a generated event from within a contract, not even the contract that was responsible for producing and emitting the event.

The keyword `event` is what Solidity uses to define events. When an event is called, its arguments are added to the blockchain once the event has completed. To begin using events, you will first need to declare them in the manner shown here:

```
event moneyTransferred(address _from, address _to, uint _
amount);
```

Then, you need to make sure that your event is being sent out from within the function:

```
emit moneyTransferred(msg.sender, _to, _amount);
```

3.4.13 Error Handling in Solidity

Solidity has a variety of routines that can be used to handle errors. In most cases, when an error occurs, the state will be reset to the way it was

before the problem occurred. Additional checks are performed to prevent unwanted access to the code.

There are a few ways to handle errors in Solidity, like `require`, `revert`, and `assert`. We will leave it to the user to explore these three error-handling mechanisms.

3.4.14 Solidity and Addresses

In this section, we will discuss how Ethereum addresses can be used and referred to within the Solidity programming language. Since the whole Ethereum ecosystem is based on transactions related to ether, it's key that we understand how different addresses work within the Solidity environment.

There are two types of accounts on the Ethereum ecosystem:

1. Externally Owned Accounts (EOA) – The private key allows for private control over any ether in the account as well as any authentication that the account requires when using smart contracts. Private keys are the distinctive data needed to produce digital signatures, which are necessary to sign documents so that money in the account can be spent.

2. Contract Accounts – There are no public or private keys connected to smart contracts, in contrast to EOAs. Smart contracts are supported by their underlying code rather than a private key. They "own themselves," as the saying goes.

   The address of each contract is obtained from the transaction that creates the contract, as a function of the account that originated the transaction and the nonce (we will cover this later). The Ethereum address of a contract can be used in a transaction

in a number of different ways, including as the receiver, for the purpose of paying payments to the contract, or for the purpose of invoking one of the functions that the contract supports.

3.4.14.1 Ethereum Address

Hash functions are an essential component in the process of address creation. To produce these, Ethereum employs the Keccak-256 hash function in the generation process.

A standard address in Ethereum and Solidity has a value length of twenty bytes (160 bits or 40 hex characters). It is equivalent to the final 20 bytes of the Keccak-256 hash that is performed on the public key. Since an address is always represented in hexadecimal format (base-16 notation), the prefix `0x` is always added to the beginning of it (defined explicitly).

The steps for creating an Ethereum address are as follows:

1. To begin, we will use the `ecparam` command provided by OpenSSL to produce an elliptic curve private key. The standard implementation of Ethereum employs the secp256k1 curve. This command will output the private key in PEM format (making use of the fantastic ASN.1 key structure), and it will be sent to the standard output.

2. Use the `openssl` command to obtain the public key from the private key.

3. To the public key, apply the Keccak-256 hashing algorithm. You should end up with a string that is a total length of 64 characters, or 32 bytes.

4. Take the least significant bytes, which are the last 40 characters or 20 bytes of the hash that was generated. (Or, to put it another way, get rid of

the first 24 characters and the first 12 bytes). The address consists of these 40 characters and 20 bytes.

When the `0x` prefix is added, the actual length of the address is increased to 42 characters. Additionally, it is essential to keep in mind that the case of the words does not matter. It is expected that all wallets would accept Ethereum addresses written in either capital letters or lowercase characters without any difference in the way they are interpreted.

The addresses on contracts are constructed in a different way. They are generated in a deterministic manner based on the following two things:

1. The address of the person who created it and sent it
2. The number of transactions that the creator has already made: nonce

The procedures involved in the formation of the address for the contract are as follows:

1. Consider the values of sender and nonce.
2. They are encoded with RLP.
3. Keccak-256 should be used to hash them.

3.4.14.2 Usage of Addresses in Solidity

To create a variable with the address data type, you must preface the variable's name with the keyword "address."

```
address sender =msg.sender
```

Here, we hold the sender address in the `sender` variable.

`msg.sender` is the built-in solidity function, which represents the address of the user invoking the functions on the smart contract.

There is one more type, called address payable, in Solidity.

In version 0.5.0 of Solidity, the difference between an address and an address payable was first introduced. The intention was to differentiate between addresses that are able to accept money and those that are not able to in this way (used for other purposes). To put it another way, a simple address is unable to receive ether but an address payable is allowed to do so.

In simple terms it's only possible to send ether to a variable that is defined as an address payable.

We are going to classify the Solidity methods that are linked with addresses into two distinct groups, or types of transactions:

1. There are three methods that are associated with ethers: `.balance()`, `.transfer()`, and `.send ()`.
2. Here are techniques associated with the interactions of contracts: `.call()`, `.delegatecall()`, and `.staticcall ()`.

3.4.14.3 Balance Method

This method returns the account balance in wei (1 ether = 10^{18} wei = 1,000,000,000,000,000,000 wei).

The technique is available to any variable that is declared as an `address.balance()`. This makes it possible to access the quantity of ether held by an externally owned account (EOA/user), as well as by a contract account. The number that was returned is the amount of ether measured in wei. Because it reads the data stored on the blockchain, a statement such as `address.balance` is considered to be a "read from the state" operation. As a result, any Solidity function that returns an `address.balance` can be declared as a view.

3.4.14.4 Transfer Function

The `address.transfer` function

- sends the quantity of ether, measured in wei, to the account that has been provided;
- reverts to the previous state and raises an exception whenever there is an error; and
- forwards a 2,300 gas stipend.

Before sending ether, the `.transfer()` method first checks the available funds at an address by applying the property balance. This takes place behind the scenes.

3.4.14.5 Contract-related Functions

Solidity provides a high-level syntax for the purpose of calling functions that are defined in other contracts (for example: `Contractx.callFunction(...)`). However, in order to use this high-level syntax, the interface of the target contract must first be understood before the compilation process begins.

`Call`, `delegatecode`, and `staticcall` are the three special opcodes that can be used by the EVM to communicate with other smart contracts. The EVM offers a total of four special opcodes that can be used for this purpose.

3.4.14.6 Gas in Ethereum

In Ethereum, we must pay for every computation, which is measured in the gas unit.

The following definition of gas is included in the Ethereum documentation:

"Gas refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network.

Since each Ethereum transaction requires computational resources to execute, each transaction requires a fee. Gas refers to the fee required to conduct a transaction on Ethereum successfully."

3.4.14.7 Ethereum Transaction Costs

Due to the high cost of computer resources and people's acute awareness of the expense of computing operations in the past (the 1960s or 1970s), people shared computers. After that, the introduction of personal computers greatly increased the accessibility of computer resources, and except in cases of severe resource limitations, people no longer give much thought to cost.

Since the Ethereum Virtual Machine serves as a common computer for the Ethereum network, we must consider how effectively we can use its resources, just as in the past. However, due to the distributed nature of its execution environment, Ethereum uses transaction fees rather than time-sharing methods.

Say, for example, we create this contract:

```
pragma solidity 0.8.13;
contract Sample {
    uint z;
    function set(uint x) public {
        z = x;
    }
    function get() public view returns (uint) {
        return z;
    }
}
```

If we want to examine the gas fee for invoking the `get` function, we can see it in Figure 3-1. This view is achieved using the remix IDE. We will cover that in length in coming chapters , so don't panic.

```
▾GASESTIMATES
{
    "Creation": {
        "codeDepositCost": "67200",
        "executionCost": "117",
        "totalCost": "67317"
    },
    "External": {
        "get()": "2437",
        "set(uint256)": "22498"
    }
}
```

Figure 3-1. *The estimate of gas for a function*

For a `set` function, the gas cost is 22498.

3.5 Summary

In this chapter, we took a look at Solidity, which is the language used on the Ethereum blockchain. We went into the details of the different constructs and syntax of the Solidity language, like types, operators, loops, and so on.

In the next chapter, we will look into the concept of wallets and how they work and operate.

CHAPTER 4

Wallets and Gateways

Users have the ability to manage their accounts on blockchain networks like Bitcoin or Ethereum through the usage of wallets. With an account on Ethereum, one can participate on the Ethereum blockchain network, such as by performing transactions on it.

An address on the Ethereum blockchain is a public string of letters and integers that begins with `0x`. Because an Ethereum address is represented by a string of numbers and letters, it is possible to check the balance of any Ethereum address on the blockchain. However, it is not possible to determine who controls any given address. Users are able to exert control over an unlimited number of addresses via wallets, which can be either software or hardware.

Users of Ethereum wallets can move their cash around within the wallet by using a private key. This key serves as the control mechanism for Ethereum wallets. Because of this, these private keys are supposed to be known only by the person who created the wallet, as anyone who knows them can access the funds in the wallet.

There is a wide variety of Ethereum wallets available, some of which may be stored on your computer or mobile device, and others of which can be stored offline by means of a piece of paper, titanium, or other hardware. You can choose the sort of Ethereum wallet that best suits your needs.

© Shashank Mohan Jain 2023
S. M. Jain, *A Brief Introduction to Web3*, https://doi.org/10.1007/978-1-4842-8975-4_4

One needs to use a secured key (known as the private key) to create a wallet. A wallet can be thought of as a store that holds the private and public keys for a user. In cryptography, a private key can be used to sign a transaction and a public key can be used to verify the signer. The idea is that the user keeps the private key secured with herself and shares the public key on the network. This allows the user to sign the transactions she wants to execute, and the verifiers using her public key can verify if she is the signer of those transactions. Wallets simplify this process of key management by providing a single place and an abstraction for clients to interact with the chain. People who want to avoid managing wallets can make use of third-party exchanges like binance that do the wallet management on the user's behalf. Since these exchanges are mostly centralized, any compromise of their servers can lead to compromise of user wallets. So, if your keys are on the exchange via an exchange-managed wallet, it is at your own risk. As the common saying goes, "Your keys, your money." From a security standpoint, it's always a better choice to have the keys off the exchange.

4.1 Types of Wallets

Some people store their Ethereum holdings in wallets designed for users, while others use cryptocurrency exchanges or other services, such as online marketplaces or loan services, offered by wallets designed for users. Wallets like these are referred to as custodial wallets, and they are distinguished from other wallets by the fact that they store users' private keys on their behalf. The user does not have direct control over the funds stored in the wallet; rather, the service manages the wallet on behalf of the user. There is a cost associated with this arrangement.

The danger that another party will not fulfill their obligations is known as counterparty risk. This risk is increased when funds are stored with a third party through the use of custodial wallets. The service that stores the private keys runs the risk of being hacked or acting maliciously, for example.

It's possible that various users would benefit from using one particular wallet over another. There are a plethora of wallets available now for the user to choose from. Most of them use Ethereum or ERC 20 standard–based tokens to access the applications on the blockchain. This access is realized by executing code in the form of a smart contract on the blockchain network. ERC 20 is a set of standards defined for creating tokens on the chain. This allows people to create their own tokens, which then can be used for their own applications if so desired.

There are different kinds of wallets available, as follows:

1. Mobile Wallet – These are mobile applications like the `bitcoin.com` app, which provides an interface for using the keys for transactions on the blockchain.

2. Desktop Wallet – This is installed as a desktop application and can be used on a laptop or home PC. An example of such a wallet is electrum.

3. Web Extensions – These are browser-based extensions that allow the wallet to be accessed from within the browser. MetaMask is an example of such wallets.

4. Hardware Wallet – These are physical hardware devices that can be used to securely manage your wallet. This is the most secure means of keeping keys secure.

In this chapter, we will discuss the MetaMask wallet, which comes as an extension to browsers like Chrome and Brave.

Once we choose a wallet, we need funds to interact with the Ethereum network. These funds are in the form of ether, which is the cryptocurrency used on the Ethereum blockchain. Ether can be purchased via exchanges like Binance or Coinbase.

All transactions on the Ethereum blockchain are validated by nodes, which are called validator nodes. They charge a fee for validating the transactions, and this fee (called gas) has to be paid by the user initiating the specific transaction. There are means via which a user can check the estimated gas fee based on estimated resources needed to execute the specific smart contract.

Since the production Ethereum network (also known as mainnet) will charge the gas fee for every transaction, it's not feasible to do development on the mainnet. To facilitate the development of decentralized applications on the Ethereum blockchain, there are many development networks available, known as testnets, that allow one to develop applications using exactly the same interfaces as the Ethereum blockchain, but without incurring real gas costs.

4.2 So, What Is a Testnet ?

When someone starts to develop a decentralized application (Dapp), they can deploy it to a test network, which from an interface perspective is the same as the main network. This test network is known as a testnet. This provides an opportunity for the developers, the community, and you to test it out before real assets are involved. Ether and tokens on a testnet are simple to acquire and have no value in the real world.

There are currently four major testnets in operation, and each functions in a manner that is analogous to the production blockchain (where your real ether and tokens reside). In most cases, projects will only be developed on a single testnet, despite the fact that individual developers may have a preference or favorite among them.

1. Ropsten – A testnet blockchain that is based on proof of work and most closely resembles Ethereum
2. Rinkeby – A blockchain based on proof of authority that was initiated by the Geth team
3. Kovan – Again, a blockchain based on proof of authority
4. Goerli – A proof of authority–based testnet

To connect our wallet to any of these testnets, we need to have these two things:

1. A wallet installed on the local machine. We will use MetaMask as the wallet here.
2. A gateway like Infura to enable our wallet connectivity to these testnets.

4.3 MetaMask

MetaMask belongs to a family of what we call HD (hierarchical deterministic) wallets. See `https://coinsutra.com/hd-wallets-deterministic-wallet/`.

4.3.1 Installation

In the next two steps, installation and configuration instructions for MetaMask are detailed for your convenience. After that, we will go through a few different configurations that you ought to become familiar with.

We can install MetaMask as an extension to Chrome/Brave. Figure 4-1 shows how the extension looks on Brave.

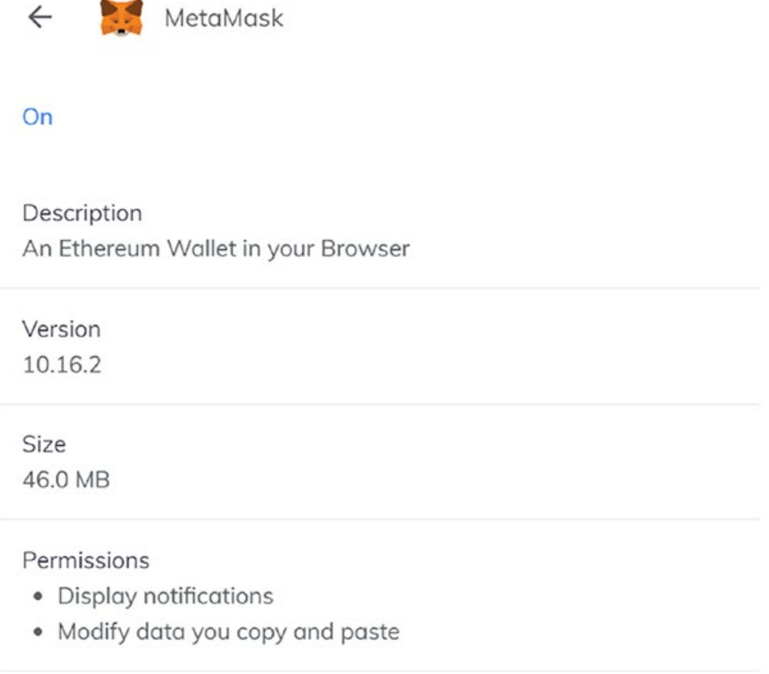

Figure 4-1. *MetaMask extension on Brave browser*

Make sure that only you can access your MetaMask account by coming up with a password and keeping it a secret from anyone else who uses the computer you share.

Immediately following the submission of your password, you will be presented with your 12-word seed phrase.

Even if they do not have the password that you chose for your account in the step before this one, anyone who knows these 12 words can log in to your account. You should never share your seed words with somebody in whom you do not have complete faith. In the event that you forget your password or something happens to your computer, you will need to re-enter these 12 words to regain access to your wallet.

Launching the MetaMask wallet extension, we see three stacked dots on the right-hand side, as shown in Figure 4-2.

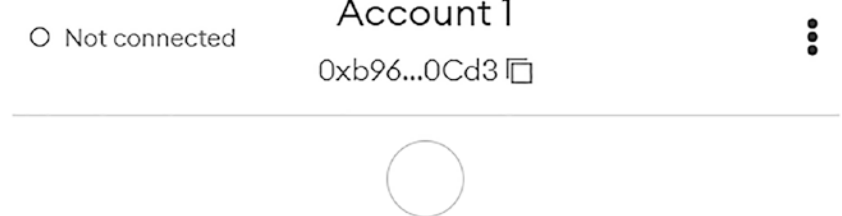

Figure 4-2. *MetaMask wallet extension*

By clicking on these three dots, we can get account details, as shown in Figure 4-3.

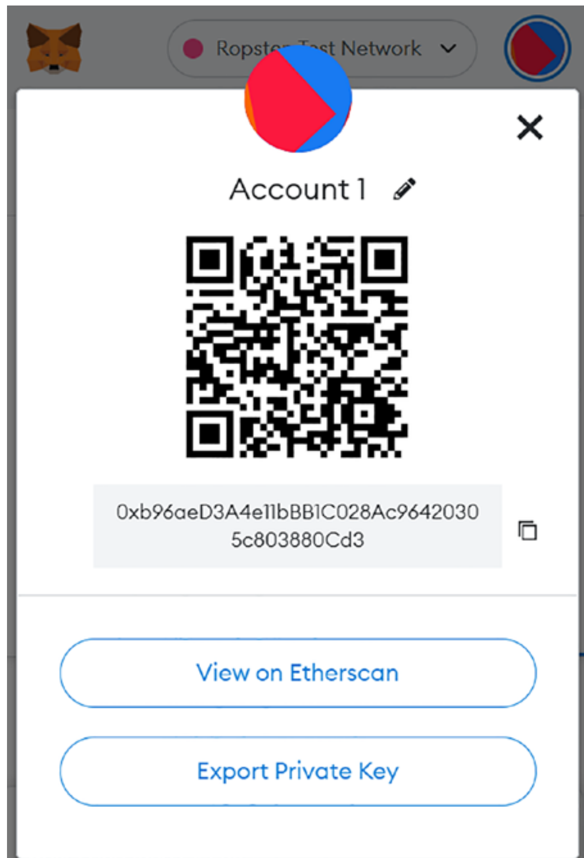

Figure 4-3. *Account details of the MetaMask wallet*

We can view the account on Etherscan by clicking on the button View on Etherscan, as shown in Figure 4-4.

Figure 4-4. *Etherscan view of the account*

Click on the dropdown menu to show the networks available, as shown in Figure 4-5.

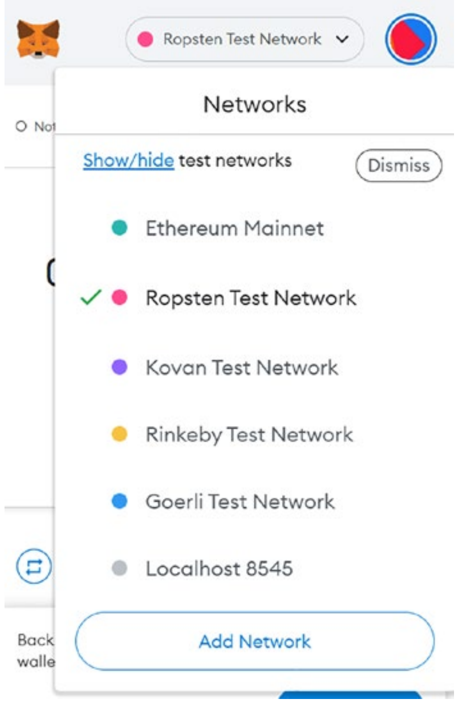

Figure 4-5. *The networks available to MetaMask wallet*

Before we add funds to the wallet from a testnet, we need to understand how the connectivity to these testnets works from MetaMask. One option is to run a testnet node ourselves and then connect our wallet to it. Another option is to go via a hosted testnet.

There are a few hosting providers for testnets. Here, we discuss one of them known as Infura.

Infura is an infrastructure-as-a-service (IaaS) and Web3 backend provider that offers a variety of services and tools to blockchain developers. This includes the application programming interface (API) suite for the Infura platform. The Infura Web3 service revolves around the flagship Infura Ethereum API as its central component. However, communication with both the InterPlanetary File System (IPFS) and Filecoin is currently

being worked on. Having said that, certain alternatives to Infura currently offer wider cross-chain connectivity than Infura itself does. Many blockchain developers are currently seeking Infura alternatives, despite the fact that Ethereum is currently the most popular programmable blockchain for the launch of decentralized applications (DApps). This occurs as Binance Smart Chain (BSC) and Polygon Network are becoming increasingly well known (previously Matic Network).

A high-level architecture of the Infura gateway is shown in Figure 4-6.

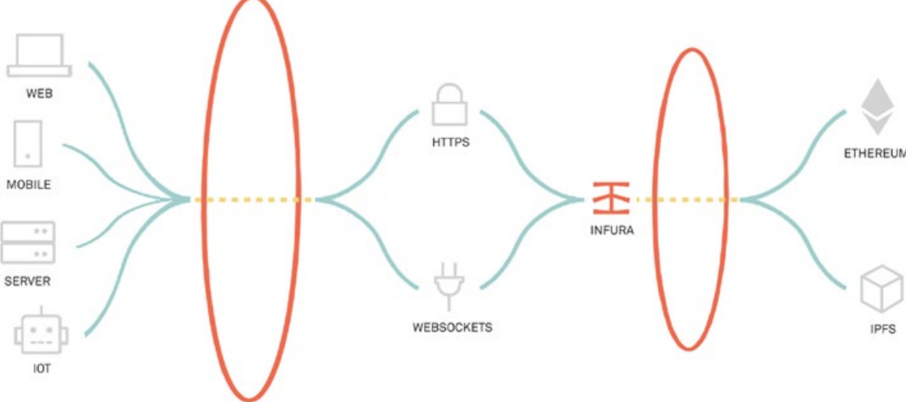

Figure 4-6. *Infura architecture*

On the left side of Figure 4-6, we see the wallet that connects to the Infura infrastructure via either https or websockets. Infura in turn provides connectivity to the blockchain networks, like Ethereum mainnet or testnets. Infura acts as a gateway for the wallets to the blockchain world.

Users who make use of the Infura Ethereum API are able to devote more of their time and resources to activities such as doing market research and product development. In addition, users are provided with a straightforward and user-friendly dashboard that allows them to obtain a greater understanding of how apps are performing. Utilizing the dashboard makes it simple to conduct application analysis and configuration. In addition, developers are able to track usage times,

the effectiveness of various request types, and a great deal more. These insights allow developers to improve their programs by gaining a deeper understanding of the people who use those applications. In addition, the Infura Ethereum API is interoperable with both testnets and mainnets, and it uses client-compatible JSON-RPC that is transferred through HTTPS and WSS. Users are also given the opportunity to obtain access to the Ethereum Archive node data that is made accessible as an add-on.

Infura is the default node provider that MetaMask utilizes, although users have the opportunity to switch to another node provider or even host their own node.

Let us add some funds to our MetaMask wallet.

We will experiment a bit with the Ropsten testnet for this example.

Navigate to `https://faucet.egorfine.com/`.

We see a screen similar to the one shown in Figure 4-7.

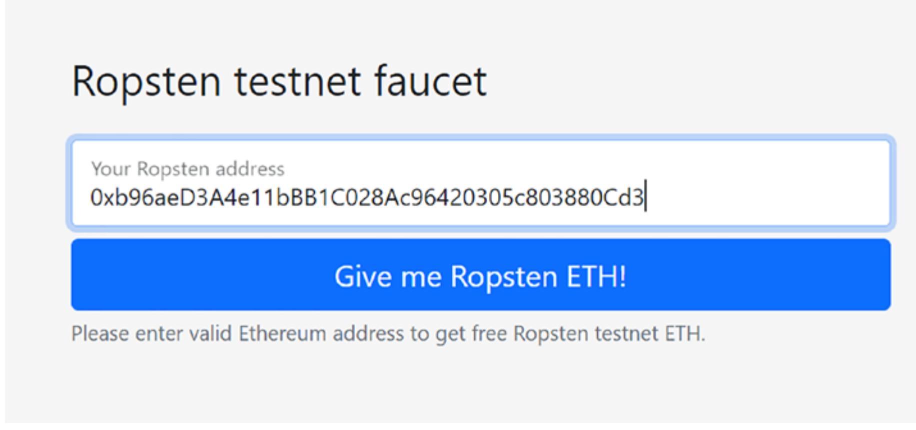

Figure 4-7. *Ropsten testnet Faucet*

In the address field, we need to put the public key that we get from the MetaMask wallet. Upon clicking the Give me Ropsten ETH! Button, we see the result shown in Figure 4-8.

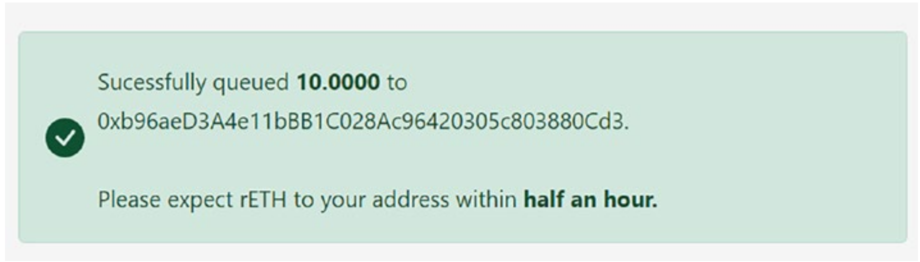

Figure 4-8. *Confirmation of ETH added to the wallet*

We can check our MetaMask wallet to see if funds got added, as shown in Figure 4-9.

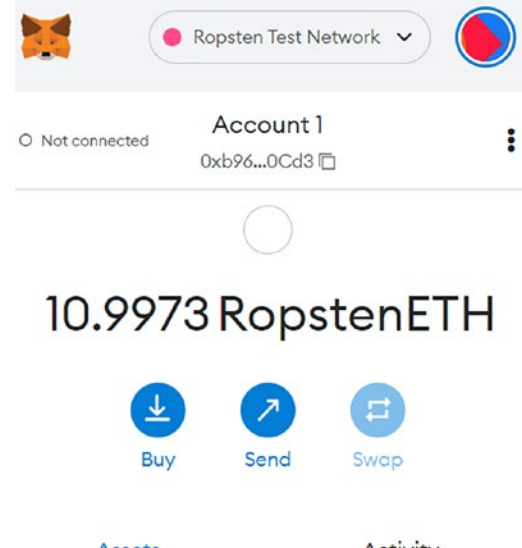

Figure 4-9. *The MetaMask wallet view. It shows the Ropsten ETH credited to the wallet*

We can see 10.9973 ETH in my wallet, out of which 0.9973 were already there from my previous addition.

We will look into Etherscan once to check the transaction. We see a screen like that shown in Figure 4-10.

Figure 4-10. *Etherscan transaction view*

We can see that 10 ether got added to my account on the Ropsten network.

We will need these funds when we create a smart contract and invoke functions on it. We already talked about gas fees in the previous chapter.

First, we will create a simple web page that uses the web3.js library to connect to the testnet via Infura. Let's now get a small introduction to web3.js.

4.4 Web3.js

The Ethereum APIs can be consumed over an HTTP-based protocol. So, one can create HTTP clients, which allows us to interface with the Ethereum networks (either mainnet or testnet). To ease the process of development, the Ethereum Foundation created a library in JavaScript that allows us to access the Ethereum blockchain programmatically. They called this library web3.js, and, as the name suggests, it's for web3-based applications, and the library itself is written in the JavaScript programming language.

This library comprises different modules, which are described in the next subsections.

4.4.1 web3-eth

A user of web3.js is able to connect with the Ethereum blockchain thanks to the web3-eth module, which offers functions that make this possible. To be more specific, these functions are able to communicate with smart contracts, accounts that are controlled by other parties, nodes, blocks that have been mined, and transactions.

Using the web3-eth library functions, one can sign the transations, can check balances in your Ethereum wallet, as well as send the signed transaction over the internet to the Ethereum blockchain.

4.4.2 web3-shh

You will be able to interact with the Whisper protocol if you make use of the web3-shh module. Whisper is a messaging protocol that was developed to facilitate the easy broadcasting of messages and the low-level, asynchronous transmission of data. The following are two instances that illustrate the point:

1. The network receives a Whisper message when `web3.shh.post` is called.
2. Apart from just sending messages, one can subscribe to messages using the `web3.ssh.subscribe` method. This allows the user to receive messages from the network.

4.4.3 web3-bzz

It's in the user's interest to have clarity as to what kind of data is stored on the blockchain. Generally, we store only transactional data on the chain. Other data, like documents, images, videos, and so on, are not stored on the blockchain. We can use decentralized storage services like Swarm, ipfs,

and so forth for those needs. We can store references to such content on the blockchain though. And this is where the module web3-bzz helps. It provides us a library-based abstraction to communicate with Swarm.

We can upload and download images, documents, videos, and audio clips to the Swarm network using the `web3.bzz.upload` and `web3.bzz.download` methods.

4.4.4 web3-net

You will be able to interact with the network attributes of an Ethereum node if you make use of the web3-net module. You will be able to get information about the node by using the web3-net module. This module allows us to extract metadata about the Ethereum node itself. As an example, the network ID can be obtained by calling `web3.net.getID`, and using `web3.net.peerCount` will return the number of peer nodes connected to a specific node.

4.4.5 web3-utils

The web3-utils module allows us to make use of some of the utility functions defined inside this library module. Included in web3-utils is a collection of utility functions that can search databases, convert numbers, and check to see whether a value satisfies a given criterion. The following are three instances that illustrate the point:

1. `web3.utils.toWei` is a converter that goes from wei to ether.
2. `web3.utils.hex` converts a hexadecimal value to a string with the `ToNumberString` function.

3. The web3.utils.isAddress function determines whether or not the given string represents a valid Ethereum address.

4.5 Infura Setup

Since we use Infura as the gateway, we need to configure the Infura endpoint, which can then be used from our applications.

In order for us to access the Infura network, the first thing that has to be done is to sign up for an Infura account and obtain an API key. We can go to `https://infura.io` to access Infura.

Please visit the Infura website to create a new account for yourself. When you open the account creation page, you will see the screen shown in Figure 4-11.

Figure 4-11. *Infura account creation page*

The next step is to go to the dashboard page and click the Create New Project button. Give your project a name and click the Create button, as shown in Figure 4-12.

CREATE NEW PROJECT

NAME

CANCEL CREATE

Figure 4-12. *Create New Project screen*

I created a project named `trial`, as shown in Figure 4-13.

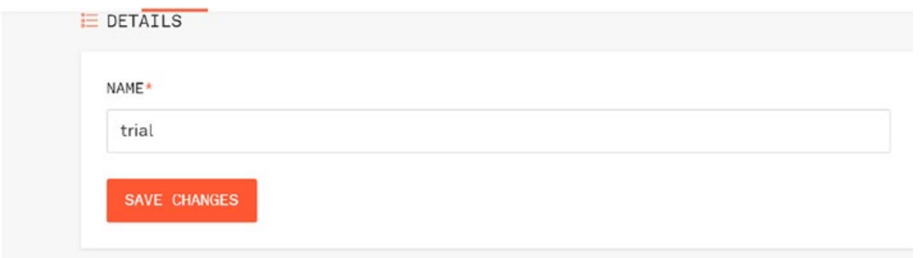

Figure 4-13. *Creating a project named trial*

We next get the project ID and project secret, as shown in Figure 4-14.

KEYS

PROJECT ID
b1c76cbfffd24d09aa70726f91de1004

PROJECT SECRET

ENDPOINTS ROPSTEN

https://ropsten.infura.io/v3/b1c76cbfffd24d09aa70726f91de1004

wss://ropsten.infura.io/ws/v3/b1c76cbfffd24d09aa70726f91de1004

Figure 4-14. *Keys for the Infura project we created*

Since I am using the Ropsten network, I choose the endpoint as Ropsten.

We get two endpoints:

1. HTTPS based
2. Secure web socket

4.5.1 Interfacing with Ropsten Network via Infura Gateway

Please copy the HTTPS endpoint URL. Remember this URL constitutes your project ID. We will use this URL in our HTML page, which we will create next.

```
Create a directory web3
Cd to web3 directory
```

Inside the web3 directory, create an HTML page. Copy the following content to this HTML page:

```
<html>
  <header>
```

```
  <title>Sample Infura connectivity check</title>

  <script
//importing the web3.js library
  src="https://cdn.jsdelivr.net/gh/ethereum/web3.js@1.0.0-
beta.36/dist/web3.min.js"  integrity="sha256-nWBTbvxhJgjslRyuAK
JHK+XcZPlCnmIAAMixz6EefVk=" crossorigin="anonymous"></script>

  <script  src="https://code.jquery.com/jquery-3.4.1.min.js"
integrity="sha256-CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo="
crossorigin="anonymous"></script>

  <script>

  if (typeof web3 !== 'undefined') {

  web3 = new Web3(web3.currentProvider);

  } else {

  // Set the provider  you want from  Web3.providers
//see the ropsten url is the one we copied from infura
  web3 =  new Web3(new Web3.providers.HttpProvider("https://
ropsten.infura.io/v3/b1c76cbfffd24d09aa70726f91de1004"));

  }

  </script>

  </header>
  <body>

  <div>
//get the last block on the ropstentestnet
  <h2>Latest Block</h2><span id="lastblock"></span>

  </div>
```

```
  <script>

  web3.eth.getBlockNumber(function (err, res) { if (err)
console.log(err)

  $( "#lastblock" ).text(res)

  })

  </script>

  </body>

  </html>
```

We get the following output on loading this page in the browser; we see the response as shown in Figure 4-15.

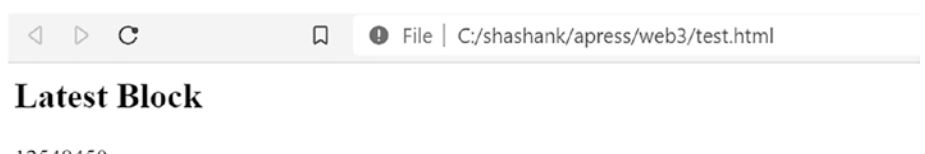

Figure 4-15. *Getting the latest block via web3 library*

This means that our test succeeded in connecting to the Ropsten testnet via the Infura gateway.

We modify the following HTML file to extract more information, like the ETH balance as well as the node information:

```
<html>

  <header>

  <title>Sample Infura connectivity check</title>

  <script
```

```
  src="https://cdn.jsdelivr.net/gh/ethereum/web3.js@1.0.0-
beta.36/dist/web3.min.js"  integrity="sha256-nWBTbvxhJgjslRyuAK
JHK+XcZPlCnmIAAMixz6EefVk=" crossorigin="anonymous"></script>

  <script  src="https://code.jquery.com/jquery-3.4.1.min.js"
integrity="sha256-CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo="
crossorigin="anonymous"></script>

  <script>

  if (typeof web3 !== 'undefined') {

  web3 = new Web3(web3.currentProvider);

  } else {

  // Set the provider  you want from  Web3.providers

  web3 =  new Web3(new Web3.providers.HttpProvider("https://
ropsten.infura.io/v3/b1c76cbfffd24d09aa70726f91de1004"));

  }

  </script>

  </header>
  <body>

  <div>

  <h2>Latest Block</h2><span id="lastblock"></span>
  <h2>My balance</h2><span id="balance"></span>
  <h2>node information</h2><span id="nodeInfo"></span>

  </div>

  <script>
```

```
web3.eth.getBlockNumber(function (err, res) { if (err)
console.log(err)

$( "#lastblock" ).text(res)

})
web3.eth.getBalance("0xb96aeD3A4e11bBB1C028Ac96420305c803880
Cd3", function  (err, res) { if (err)  console.log(err)

$( "#balance" ).text(res)


})


web3.eth.getNodeInfo(function (err, res) { if (err) console.
log(err)


$( "#nodeInfo" ).text(res)

})

</script>

</body>

</html>
```

Loading this in a browser gives the output shown in Figure 4-16.

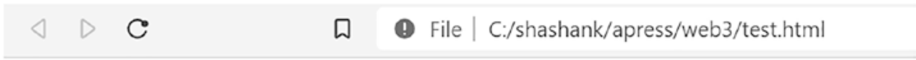

Figure 4-16. *Showing block information and balance information*

The same functionality we showcased via the browser can be achieved using nodejs as the JavaScript-based application server.

4.6 Summary

In this chapter, we looked at different kinds of wallets and how they work. We detailed how MetaMask wallet works. We also looked at how gateways work and their purpose in the web3 world.

In the next chapter, we will look into how we can use the Remix IDE (a browser-based environment) to compile and deploy smart contracts.

CHAPTER 5

Introduction to Remix IDE

To develop smart contracts and then manage the lifecycle of those contracts—like compilation, testing, deployment, and updates—we need an integrated development environment (IDE). There are many options available. Remix IDE is a web-based IDE (which means it does not require any software installation). Remix comes with a full suite of development and deployment tools integrated for developing and managing the lifecycle of smart contracts.

In this chapter, we will introduce you to the Remix IDE and see how it can help with creating and managing the lifecycle of smart contracts.

5.1 Remix IDE

Remix IDE provides a browser-based environment for creating, compiling, testing, and deploying Ethereum-based smart contracts on the blockchain network. Actually, working with Remix IDE is a piece of cake! In this chapter, we will go through how to make use of it.

The Remix IDE is laid out into different panels, each having a different purpose, as shown in Figure 5-1.

© Shashank Mohan Jain 2023
S. M. Jain, *A Brief Introduction to Web3*, https://doi.org/10.1007/978-1-4842-8975-4_5

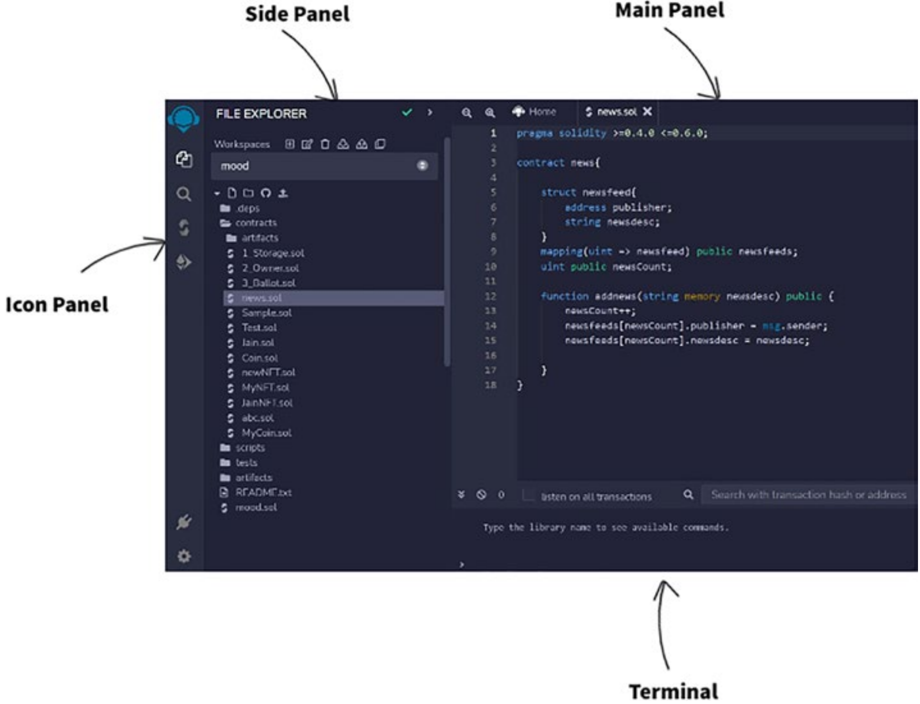

Figure 5-1. *Showing the different parts of the Remix IDE*

1. Icon Panel – This shows the different icons for functions like compiling, running, and deploying smart contracts.

2. Side Panel – This shows the File Explorer where we can create Solidity-based smart contracts.

3. Main Panel – This is specific to editing the files. We will show examples later in the chapter.

4. Terminal – This is where you see the results of the execution of various commands. You can run custom scripts directly from the terminal.

We will go over each component that makes up the Remix IDE. We will create a straightforward smart contract, then proceed to compile and deploy it on the Ropsten test network using MetaMask. Don't worry: We will walk you through the process of writing your smart contract.

The smart contract we create has two simple functions:

1. Set a message of your choice. This will store the message on the Ropsten network.
2. Retrieve the message.

We need an understanding of Solidity and MetaMask alongside the Remix IDE to create these smart contracts. We already learned how Solidity and MetaMask work in the previous chapters. So let's get started.

Any smart contract creation done via the Remix IDE has at least three essential steps:

1. Write the smart contract using Solidity.
2. Compile the contract.
3. Deploy the contract.

To create the contract, navigate to the Remix IDE at https://remix.ethereum.org/, as shown in Figure 5-2.

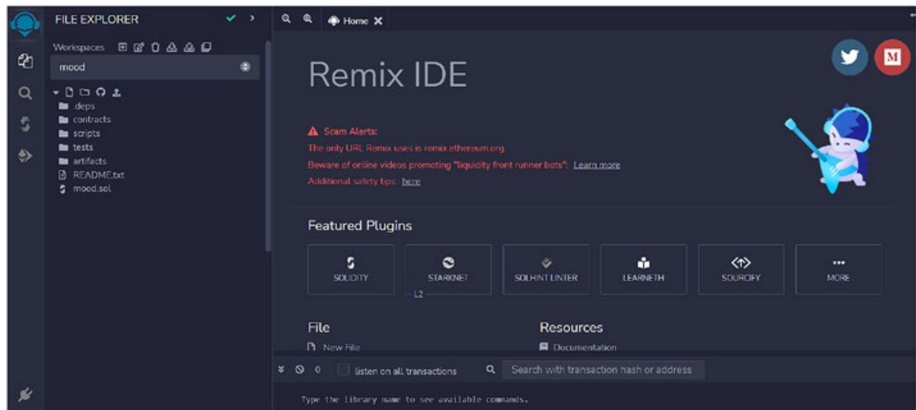

Figure 5-2. *Homepage of Remix IDE*

Under the default workspace, we see a folder for contracts. Right-click on it, and then click on Create New File.

I created a file named Test.sol, as shown in Figure 5-3.

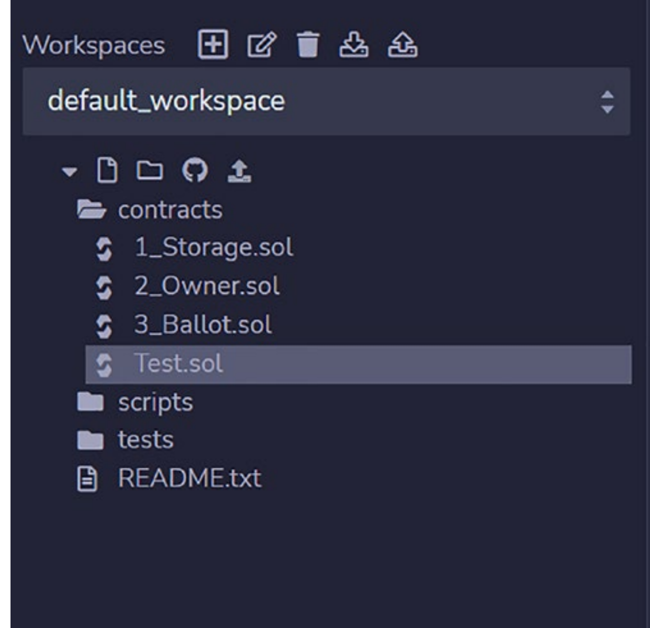

Figure 5-3. *Creating a new file called Test.sol (.sol is the solidity extension)*

Copy the code from Listing 5-1 into the right-side window.

Listing 5-1. Code for simple smart contract

```
pragma solidity 0.8.5;

contract Test{
    // state variable for holding the message
    string private message;

    // Initialize the message to Hello!!.
    constructor() public {
        message = "Welcome";
    }
```

```
    /** @dev Function to set a new message.
      * @param newMessage The new message.
      */
    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

    /** @dev Function to return the message.
      * @return The message string.
      */
    function getMessage() public view returns (string memory) {
        return message;
    }
}
```

As we can see, this creates a contract by name of Test in the Solidity language. The contract has two functions:

1. `setMessage`
2. `getMessage`

Now it's time to compile the contract, as shown in Figure 5-4.

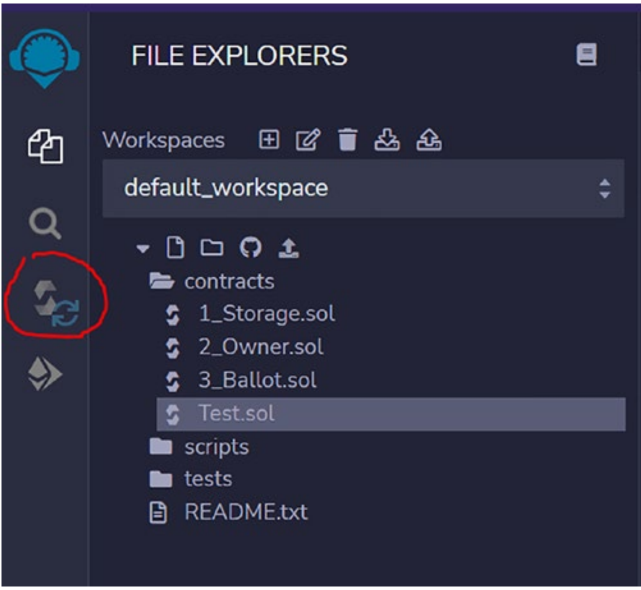

Figure 5-4. *Compile the Test.sol file by clicking on the Compile button, shown circled in red*

Click on the button highlighted in red. We will see the screen shown in Figure 5-5.

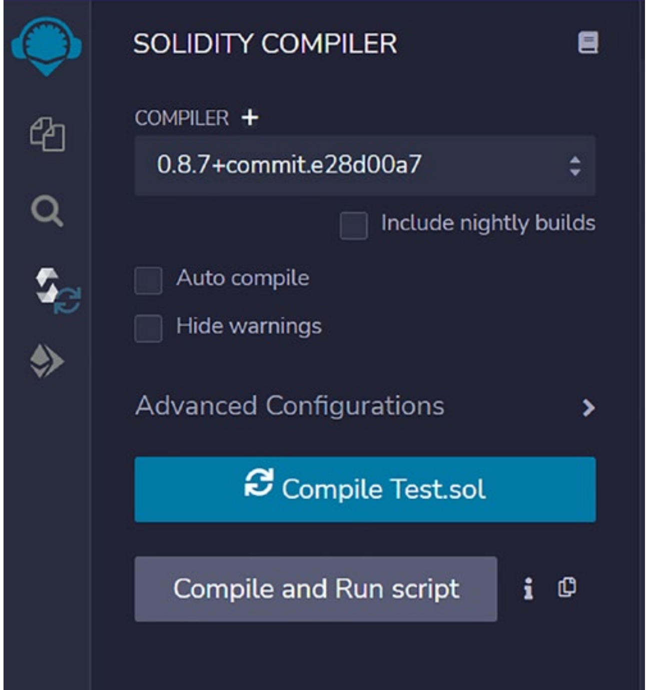

Figure 5-5. *Compilation screen for smart contract*

Click on the Compile Test.sol button, and the screen shown in Figure 5-6 will appear.

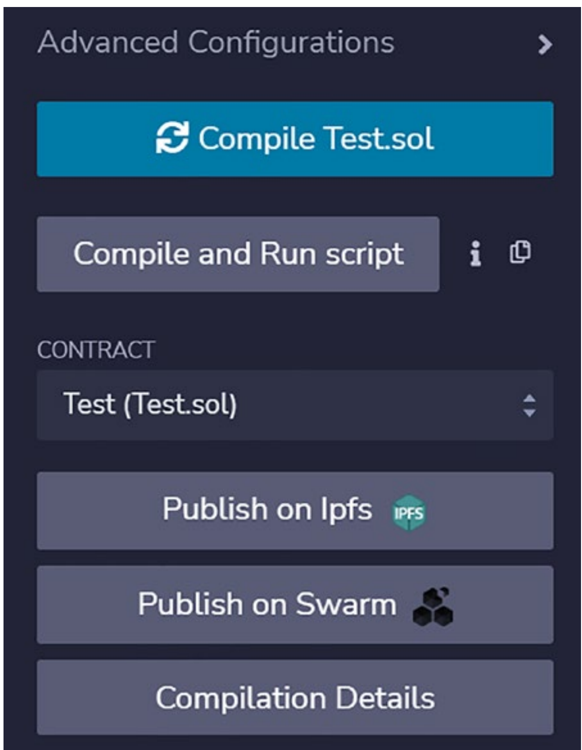

Figure 5-6. *Compiled Test.sol*

Click on the Compilation Details button to explore the different aspects of the contract and environment, as shown in Figure 5-7.

Figure 5-7. *See compiler version and language*

Once we have compiled the smart contract, we will deploy it. Click on the button highlighted in red, as shown in Figure 5-8.

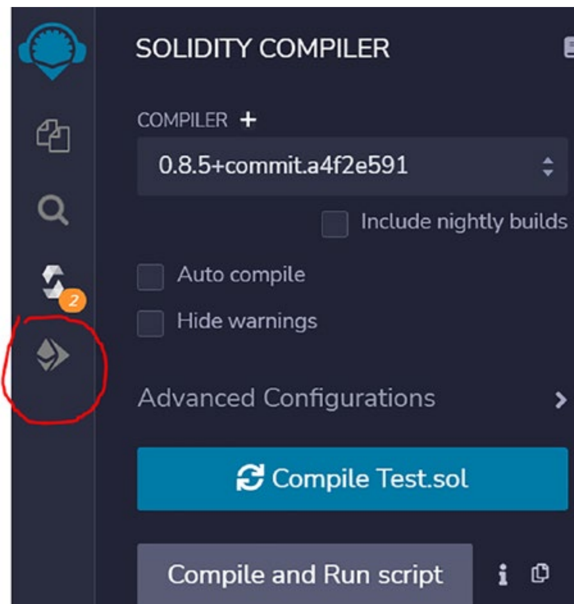

Figure 5-8. *Deployment button marked via red circle*

Once the Deploy button has been clicked, we will see the screen shown in Figure 5-9.

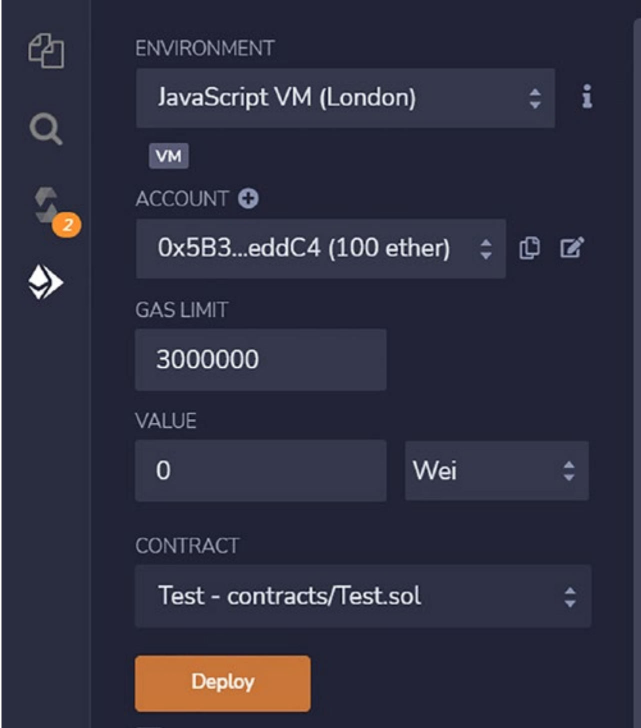

Figure 5-9. *Environment to be chosen for deployment*

We can choose from multiple deployment environments, as shown in Figure 5-10.

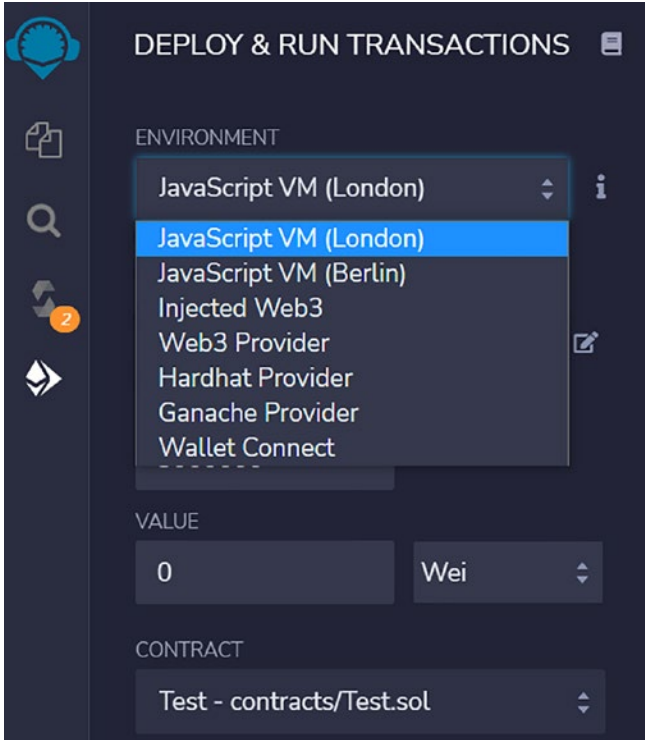

Figure 5-10. *Different environments for deployment of smart contract*

In our case, we need to choose the Injected Web3 environment, which will connect us to the MetaMask wallet for transaction signing and gas purposes.

The moment we choose the Injected Web3 environment, we will see the Ropsten network, since this is the network we have chosen for our MetaMask wallet. This is shown in Figure 5-11.

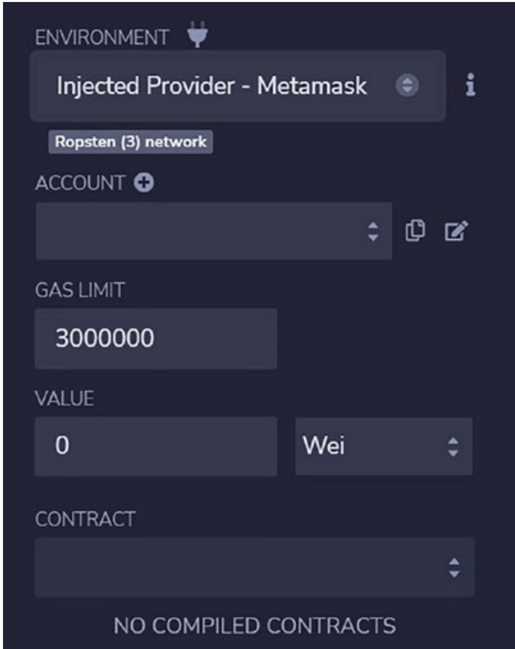

Figure 5-11. *We choose the Ropsten test network for deployment*

We see two other display sections in Figure 5-11. "Account" refers to the account we created using MetaMask, and "Gas Limit" is the maximum gas allowed for transactions performed via Remix.

Now, let's go ahead and click the Deploy button. This will open the MetaMask wallet (we have deployed MetaMask as a Brave browser extension), as shown in Figure 5-12.

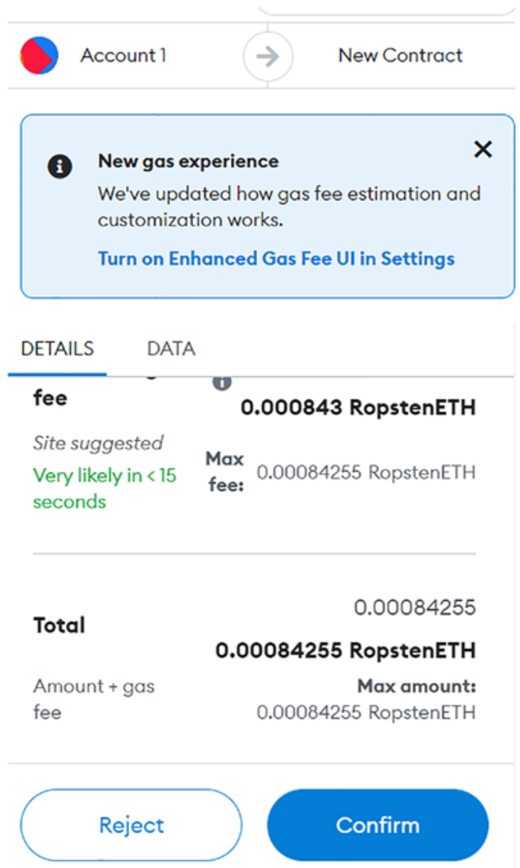

Figure 5-12. *MetaMask wallet opens up to confirm payment of gas for deployment*

We can see the amount of the gas fee to be paid for this transaction in terms of ether.

We will confirm this transaction to get it published on the Ropsten testnet.

We can check on Etherscan to see the state of the transaction, as shown in Figure 5-13.

[This is a Ropsten **Testnet** transaction only]

Transaction Hash:	0x91383171f1536aa36674684edf9c54a0d95e521f5539b0de2711e254664fd341
Status:	Pending
Block:	(Pending)
Time Last Seen:	00 days 00 hr 00 min 19 secs ago (Jul-08-2022 05:34:49 AM)
From:	0xb96aed3a4e11bbb1c028ac96420305c803880cd3

Figure 5-13. *Etherscan view of the deployment transaction*

As we can see, initially it's in a Pending state. Also, one can see the key in the From field is the public key of my account. We can also validate the key by opening the account details on MetaMask, as shown in Figure 5-14.

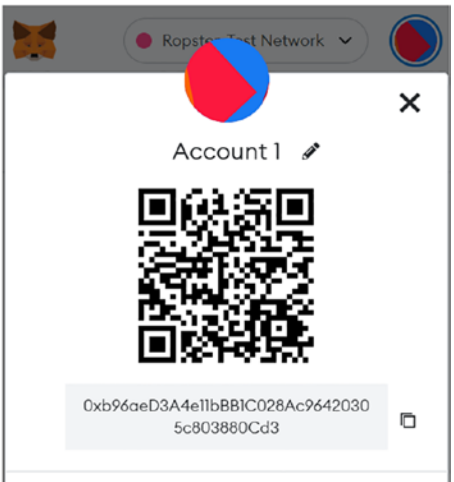

Figure 5-14. *MetaMask wallet showing the public key, which we can confirm on Etherscan*

After few seconds, the transaction gets confirmed on the Ropsten testnet. The details are shown in Figure 5-15.

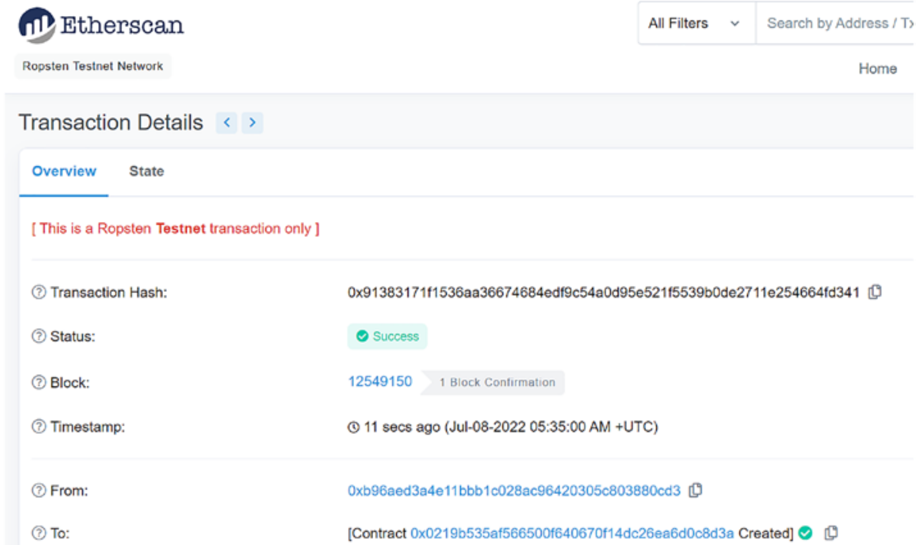

Figure 5-15. *Etherscan view of the deployment transaction (completed state)*

We also see now a To field. This To field is the address of the contract. Remember that in Chapter 3 we discussed two types of addresses. This is the contract address.

We can also check this in Remix IDE, as shown in Figure 5-16.

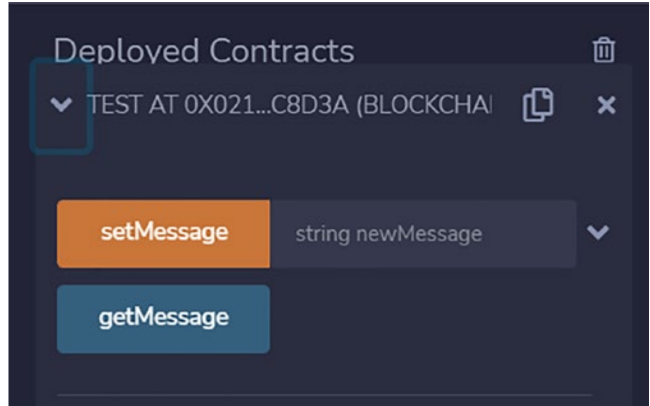

Figure 5-16. *Contract functions on Remix IDE*

Apart from the contract address, Remix IDE also provides a way to invoke these contracts.

Let's go ahead and invoke the `setMessage` function on the smart contract we just deployed.

On setting the message in the text box we will see the view shown in Figure 5-17.

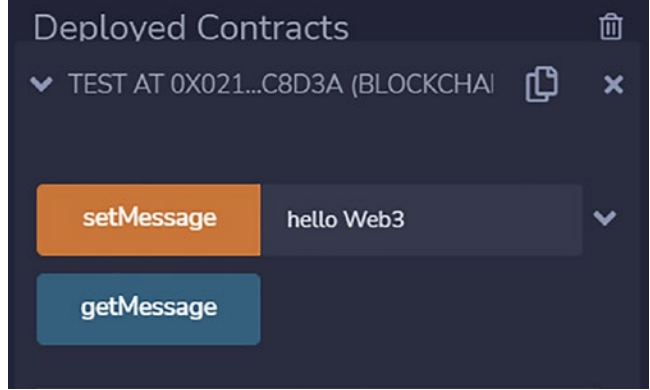

Figure 5-17. *Feeding the message for invoking a smart contract function*

And after we click the setMessage button, the MetaMask wallet will open again. This will allow us to sign the transaction and pay the gas fee.

We can see the amount and so forth in Figure 5-18.

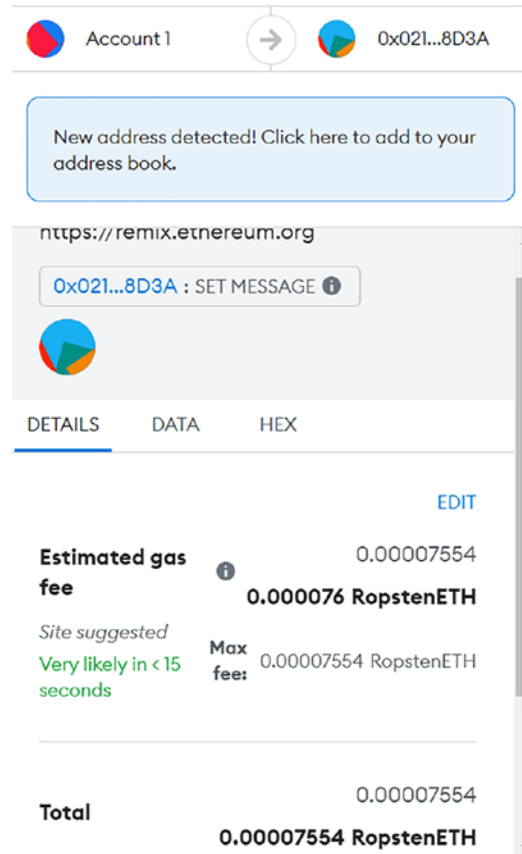

Figure 5-18. *Gas consumed for invoking the setMessage function on the smart contract*

We will confirm the transaction in MetaMask now. And we can see the transaction details on Etherscan, as shown in Figure 5-19.

Transaction Hash:	0x6fa950c713c167d034cc2b7ab32bb4da31ad91ed2482bd2106b4ab7e22d544cf
Status:	Success
Block:	12549195 3 Block Confirmations
Timestamp:	39 secs ago (Jul-08-2022 05:45:12 AM +UTC)
From:	0xb96aed3a4e11bbb1c028ac96420305c803880cd3
To:	Contract 0x0219b535af566500f640670f14dc26ea6d0c8d3a
Value:	0 Ether ($0.00)
Transaction Fee:	0.000075505000211414 Ether ($0.00)
Gas Price:	0.000000002500000007 Ether (2.500000007 Gwei)

Figure 5-19. *Etherscan view of the smart contract function invocation transaction*

There are a few things one should remember about the numbers just shown.

The quantity of gas that is used for a transaction is measured in gas units consumed per transaction. This quantity, in turn, is a measure of how complicated the transaction was. This is dependent on the number of operations and amount of storage space being used by the smart contract.

The price that you are willing to pay for one gas unit is referred to as the price per gas unit. Your transaction will be processed at a different speed as a result of this. This process is referred to as a Priority Gas Auction (PGA), and it means that all transactions are participating in an auction to determine which miners will have the opportunity to include their transactions in the blocks that are about to be mined.

The amount of ETH that goes toward covering the transaction fee is determined by using the following formula:

```
Transaction Fee = Gas Units Used * Price per Gas Unit
```

Calling `getMessage` by clicking the getMessage button gives us the following output in the Remix IDE console:

```
{
        "0": "string: hello Web3"
}
```

This is what we had set as the message on the chain.

5.2 Creating Own Token

Now, with knowledge of Solidity, MetaMask, and Remix, we move on to a more concrete example of an application on the Ethereum blockchain.

How many times you might have wondered and wished to have your own cryptocurrency. With the development of the ERC-20 standard, creating your own coin is not that hard a job. We will go into the details in this section.

The number 20 serves as the proposal's identifier, and ERC is an abbreviation for Ethereum's Request for Comments. The ETH network was targeted for improvement when ERC-20 was developed.

ERC-20 is the standard for creating tokens for DApps on the Ethereum blockchain. All Ethereum-based tokens are required to adhere to the ERC-20 standard, which provides a set of rules for creating these tokens.

Tokens, as defined by ERC-20, are assets based on blockchains that may be sent and received and have value attached to them. To a significant extent, ERC-20 tokens are analogous to cryptocurrencies such as Bitcoin and Litecoin, with the difference that ERC-20 tokens operate only on the Ethereum blockchain network.

Prior to the development of the ERC-20 standard, anyone who wanted to produce their own token was forced to start from scratch, which resulted in a wide variety of distinct tokens. This was the case because there were no specific guidelines or structures for developing new tokens. Adding new

types of tokens necessitated that the developers of wallets and exchange platforms read through the source code of each individual token and gain an understanding of it before they could begin to work with those tokens on their respective platforms. This was a particularly arduous process. It goes without saying that it was quite challenging to incorporate new tokens into any program. Wallets can incorporate ERC-20–based tokens into their platform and allow usage of these tokens via the apps. The ERC-20 token standard has made it virtually simple and seamless to interface between different tokens.

The standard specifies nine different functions that must be implemented by a smart contract, along with three that can be implemented if desired, including the following:

- `totalSupply` – This function specifies the total supply for this token. Once the maximum capacity is reached, no more tokens can be generated.
- `balanceOf` – This function, when invoked, returns the balance for a specific wallet. We will show how this can be invoked from Remix IDE.
- `transfer` – This function moves tokens to a specific address. The tokens are deducted from the sender address in this case.
- `transferFrom` – This function transfers tokens from one user address to another. Here, the available tokens in supply remain the same, but it’s a transfer between two accounts.
- `approve` – This function determines, taking into account the overall quantity of tokens, whether or not it is permissible for a smart contract to allot a specific number of tokens to a user.

- `allowance` – This function checks whether the transferor address has enough tokens to transfer to the transferee.

So, in simple terms, ERC-20 provides the specifications or interface that allows one to create tokens on the Ethereum blockchain.

Open the Remix IDE and create a file under the Contract directory and name it Jain.sol.

Copy the code from Listing 5-2 to the file Jain.sol.

Listing 5-2. Code for creating own token based on ERC-20 specs

```
// this is the ERC-20 interface which we will implement to
create our own coin
pragma solidity 0.8.5;
interface ERC20Interface {
    function totalSupply() external view returns (uint);
    function balanceOf(address tokenOwner) external view
    returns (uint balance);
    function allowance(address tokenOwner, address spender)
    external view returns (uint remaining);
    function transfer(address to, uint tokens) external returns
    (bool success);
    function approve(address spender, uint tokens) external
    returns (bool success);
    function transferFrom(address from, address to, uint
    tokens) external returns (bool success);

    event Transfer(address indexed from, address indexed to,
    uint tokens);
    event Approval(address indexed tokenOwner, address indexed
    spender, uint tokens);
}
```

```
// implementation code for ERC20 interface
//here we create a smart contract named JainToken
contract JainToken is ERC20Interface {
    string public myTokenSymbol;
    string public  myTokenName;
    uint8 public tokenDecimals;
    uint public _totalSupplyOfToken;

    mapping(address => uint) tokenBalances;
    mapping(address => mapping(address => uint)) allowed;

 //this is where we initialize our token with total supply,
name etc
    constructor() public {
        myTokenSymbol = "JAIN";
        myTokenName = "Shashank Jain Coin";
        tokenDecimals = 2;
        _totalSupplyOfToken = 200000;
        tokenBalances[msg.sender] = _totalSupplyOfToken;
        emit Transfer(address(0), msg.sender, _
        totalSupplyOfToken);
    }
 // function to return the supply at any point in time.
    function totalSupply() public override view returns
    (uint) {
        return _totalSupplyOfToken  - tokenBalances
        [address(0)];
    }
 //function to check balance of tokens at a specific address
    function balanceOf(address tokenOwner) public override view
    returns (uint balance) {
        return tokenBalances[tokenOwner];
```

```
    }
 // function for transferring tokens to a specific address.
    function transfer(address to, uint tokens) public override
    returns (bool success) {
//checks if there is enough balance in the sender address
        require(tokens <= tokenBalances[msg.sender]);
//deduct tokens from the sender
        tokenBalances[msg.sender] = tokenBalances[msg.
        sender]-tokens;
        // add tokens to the recipient address
        tokenBalances[to] = tokenBalances[to] + tokens;
// once transfer is done emit a message
        emit Transfer(msg.sender, to, tokens);
        return true;
    }

    function approve(address spender, uint tokens) public
    override returns (bool success) {
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }
 // this is same as approve but here we can specify from
address which can be different from //message sender
        function transferFrom(address from, address to, uint
        tokens) public override returns (bool success) {

require(tokens <= tokenBalances[from]);
        tokenBalances[from] = tokenBalances[from]-tokens;

        require(tokens <= allowed[from][msg.sender]);
        allowed[from][msg.sender] = allowed[from][msg.
        sender]-tokens;
```

```
        uint c=0;
        c = tokenBalances[to] + tokens;
        require(c >=tokenBalances[to] );

        emit Transfer(from, to, tokens);
        return true;
    }
 // checks if tokenOwner is allowed to make the transfer
    function allowance(address tokenOwner, address
    spender)  public override view returns (uint remaining) {
        return allowed[tokenOwner][spender];
    }

    fallback() external payable {
        revert();
    }
}
```

The coin name is Shashank Jain Coin and its symbol is JAIN. We kept the supply fixed to 200000.

Compile and deploy the contract. Remember to choose Injected Web3 as the environment with which to connect to MetaMask, as shown in Figure 5-20.

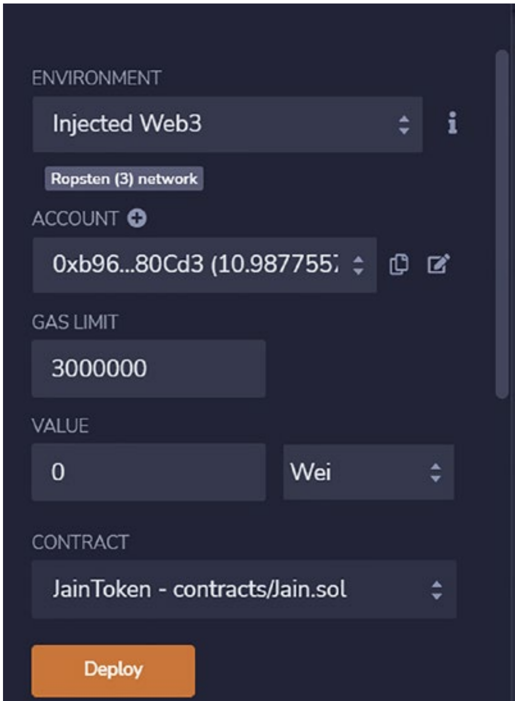

Figure 5-20. *Remix IDE showing the deployment screen*

It will ask for approval of transaction in MetaMask, as shown in Figure 5-21.

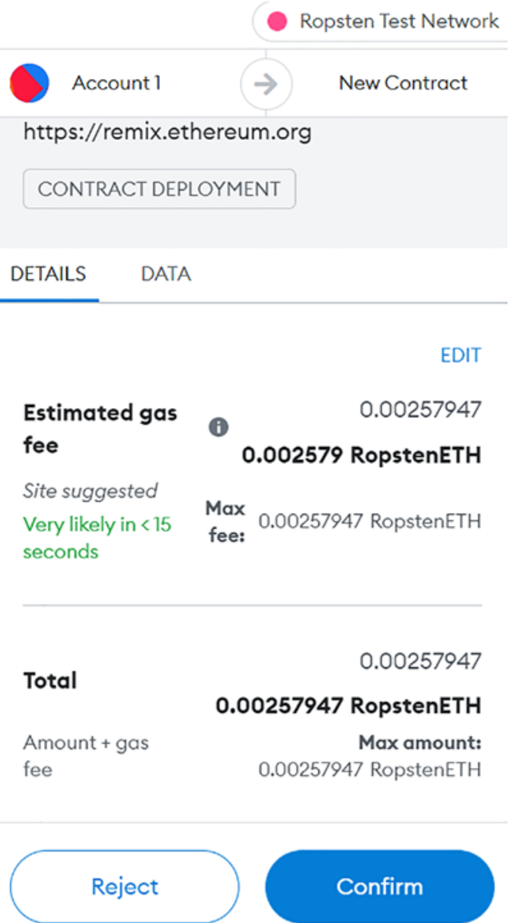

Figure 5-21. *MetaMask opens up on deployment and asks for a confirmation*

Once we confirm, we can see the transaction in Etherscan, as shown in Figure 5-22.

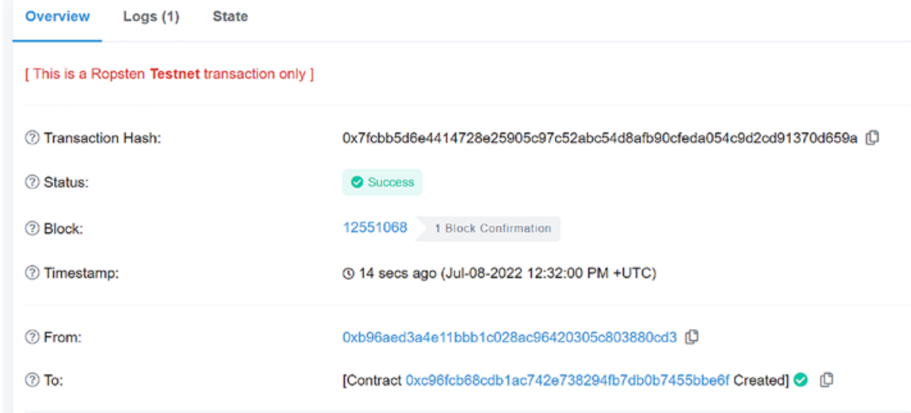

Figure 5-22. *Etherscan view of the transaction just done*

https://ropsten.etherscan.io/tx/0x7fcbb5d6e4414728e25905c97c52abc54d8afb90cfeda054c9d2cd91370d659a

Copy the contract address from the To field.

In my case it is 0xc96fcb68cdb1ac742e738294fb7db0b7455bbe6f.

Now we navigate back to the Remix IDE and open the contract, as shown in Figure 5-23.

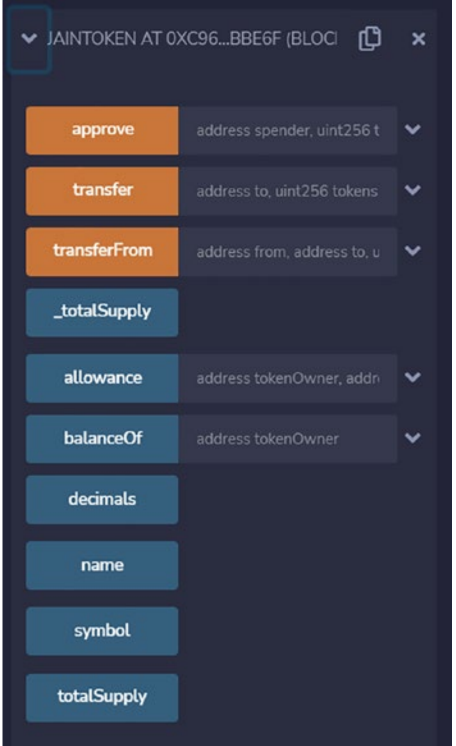

Figure 5-23. *Remix view of the deployed contract*

If you have multiple contracts deployed, please make sure you select the right contract based on the contract address.

The first function we invoke is to check the tokens balance. Since the token owner account is the account that created the contract, we will copy the account address from MetaMask. Open MetaMask, as shown in Figure 5-24.

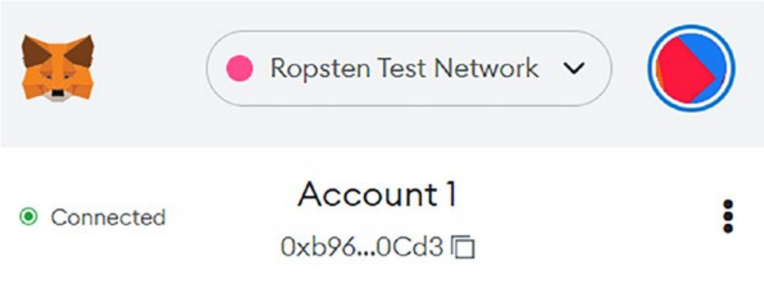

Figure 5-24. *MetaMask showing the account*

The address for the owner in my case is 0xb96aeD3A4e11bBB1C028Ac96420305c803880Cd3.

We will now check the balance in this account using the contract API in Remix IDE for the deployed contract, as shown in Figure 5-25.

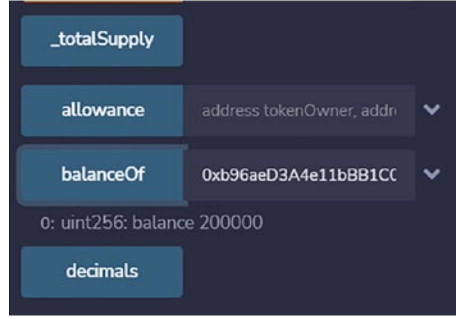

Figure 5-25. *Feeding the address to check the balance*

We can see that the tokens in this account are what we programmed it to be when we created and deployed the contract.

Next, we do a transfer of some tokens from this account to another account. I have already created another account in MetaMask with the name "test," and we can see that in Figure 5-26.

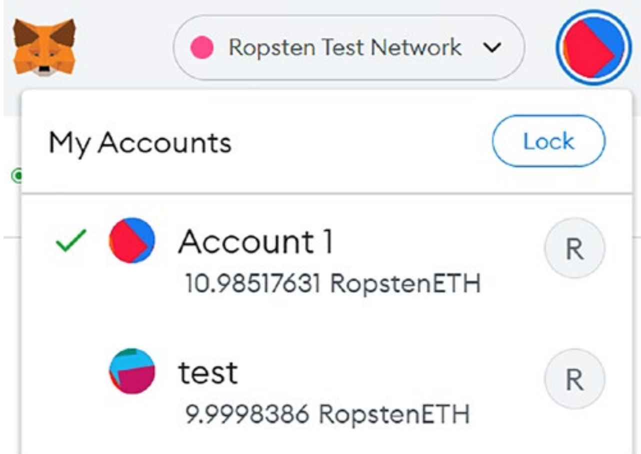

Figure 5-26. *MetaMask showing both of my accounts*

We get the address of test as 0x1A703B299d764B4e28Dc2C7849CFeDF9979D2430.

We will now transfer 100 tokens to this address using the APIs in Remix IDE.

Since we have kept decimals to two, the number of coins displayed would be tokens/10^2, which means tokens/100 in our case. So when we transfer 100 tokens we see them as 1 JAIN token. We can see the deployed contract transfer function in Figure 5-27.

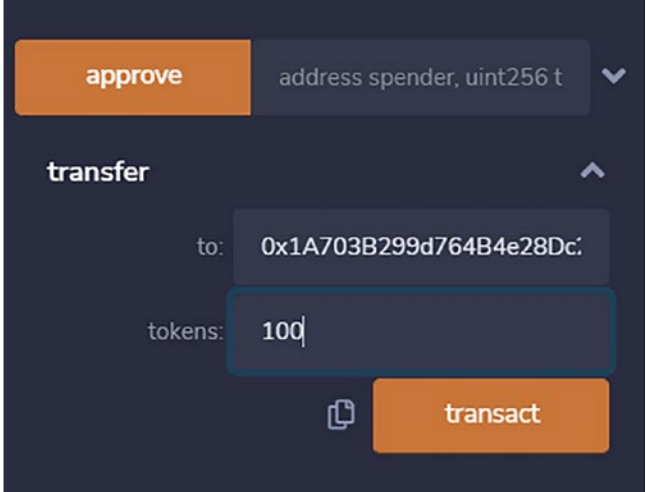

Figure 5-27. *Remix view of transfer function of the deployed contract*

Click on the Transact button.

MetaMask opens up for approval, as shown in Figure 5-28.

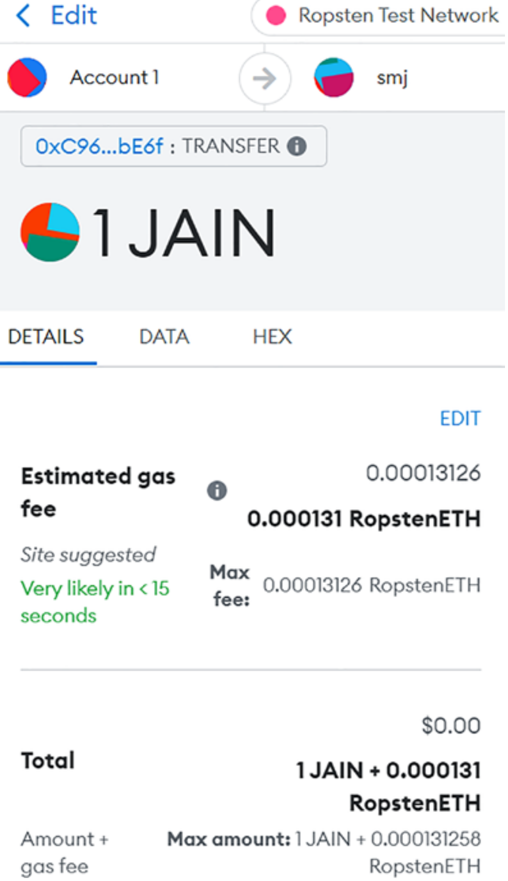

Figure 5-28. *MetaMask view of the transfer transaction*

It asks for 1 JAIN token to be transferred.

We confirm the transaction.

Now, to display our ERC token in MetaMask, click on the Assets tab, as shown in Figure 5-29.

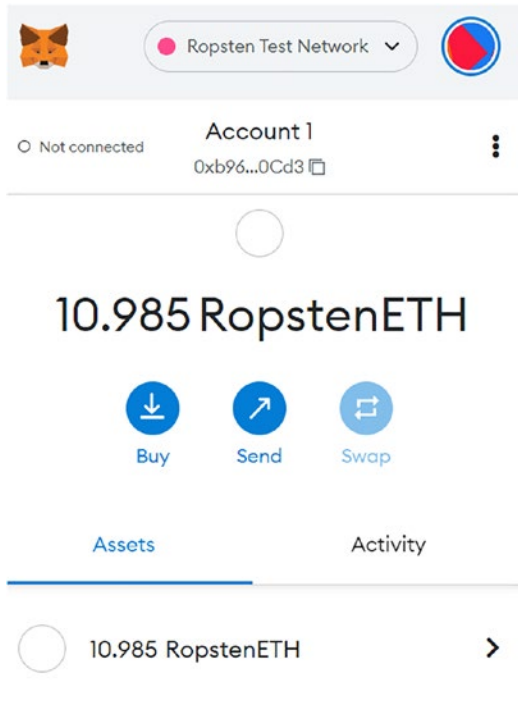

Figure 5-29. *MetaMask view of the account*

In the Assets tab, we need to click on the Import Tokens button. Paste the token contract address. Once we paste, the Token Symbol field is populated, as shown in Figure 5-30.

Import Tokens

Custom Token

Anyone can create a token, including creating fake versions of existing tokens. Learn more about scams and security risks.

Token Contract Address

0xc96fcb68cdb1ac742e738294fb7db

Token Symbol Edit

JAIN

Token Decimal

Figure 5-30. *Import of our token (JAIN token)*

Now we will see the balance of JAIN tokens, as shown in Figure 5-31.

Figure 5-31. *Shows balance of JAIN tokens in Account 1*

We will now navigate to the transferee account and check whether the JAIN token arrived or not. We will again have to import the JAIN token into the test account. On doing the import, we see the balance under the test account, as shown in Figure 5-32.

***Figure 5-32.** Shows that 1 JAIN token arrived at the test account*

We now see that the test account has 1 JAIN credited.

We go back to Remix IDE and also check the balance there, as shown in Figure 5-33.

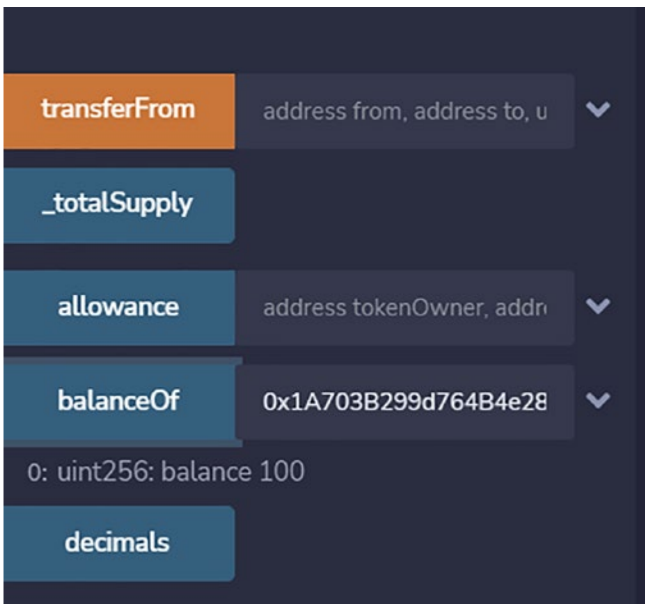

***Figure 5-33.** Balance in test account*

We see 100 as the balance in Remix IDE.

5.3 Summary

In this chapter, we introduced the reader to the concept of a Web3 app. We learned about Remix IDE as well as about a simple manifestation of a Web3 app in the form of an ERC-20 token on the Ropsten testnet using the MetaMask wallet.

In the next chapter, we will introduce the reader to Truffle, which is another development environment for DApp development.

CHAPTER 6

Truffle

So far, we have seen how to use the Remix integrated development environment (IDE) to create smart contracts, compile them, and then deploy them to the blockchain network. In our example, we deployed our contracts to the Ropsten test network. Apart from Remix, there are standalone frameworks that allow us to create, compile, test, and deploy contracts. Truffle is one such framework available to us.

6.1 Truffle Installation

Before we use Truffle, we need to install the following software. We will install everything on a Windows machine, but steps will be similar for a Linux system.

6.1.1 Installing Node

Install node.js (version 16.16.0) from the following link:

```
https://nodejs.org/en/download/
```

This will include npm 8.11.0 with it.

© Shashank Mohan Jain 2023
S. M. Jain, *A Brief Introduction to Web3*, https://doi.org/10.1007/978-1-4842-8975-4_6

6.1.2 Install Truffle

Node Package Manager (NPM) is the best way to install Truffle. Install Truffle after installing NPM on your computer by opening the Terminal and typing the following:

```
npm install --global windows-build-tools@4.0.0
npm install -g truffle
```

Once we run these commands, we will see output similar to Figure 6-1.

```
C:\Windows\system32>npm install -g truffle
npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.
npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.
npm WARN deprecated mkdirp-promise@5.0.1: This package is broken and no longer maintained. 'mkdirp' itsel
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
npm WARN deprecated multicodec@1.0.4: This module has been superseded by the multiformats module
npm WARN deprecated uuid@2.0.1: Please upgrade  to version 7 or higher.  Older versions may use Math.rand
ee https://v8.dev/blog/math-random for details.
npm WARN deprecated uuid@3.4.0: Please upgrade  to version 7 or higher.  Older versions may use Math.rand
ee https://v8.dev/blog/math-random for details.
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/is
npm WARN deprecated multibase@0.6.1: This module has been superseded by the multiformats module
npm WARN deprecated multibase@0.7.0: This module has been superseded by the multiformats module
npm WARN deprecated uuid@3.2.1: Please upgrade  to version 7 or higher.  Older versions may use Math.rand
ee https://v8.dev/blog/math-random for details.
npm WARN deprecated uuid@3.2.1: Please upgrade  to version 7 or higher.  Older versions may use Math.rand
ee https://v8.dev/blog/math-random for details.
npm WARN deprecated uuid@3.3.2: Please upgrade  to version 7 or higher.  Older versions may use Math.rand
ee https://v8.dev/blog/math-random for details.
npm WARN deprecated multicodec@0.5.7: This module has been superseded by the multiformats module
npm WARN deprecated node-pre-gyp@0.11.0: Please upgrade to @mapbox/node-pre-gyp: the non-scoped node-pre-
recieve updates in the future
npm WARN deprecated cids@0.7.5: This module has been superseded by the multiformats module

added 573 packages, and audited 574 packages in 5m
```

Figure 6-1. *Truffle installation*

Install the dotenv package to allow you to use a .env file to store private environment variables on your local machine.

```
npm install dotenv
```

Install dependencies for HD Wallet next:

```
npm i @truffle/hdwallet-provider@next
```

6.2 Smart Contract Deployment via Truffle

Make a new folder for your project by issuing the following command:

```
mkdir mycoin
```

Go into the new directory by issuing the following command:

```
cd mycoin
```

Initialize a Truffle project by issuing the following command:

```
truffle init
```

This will create the following directories:

- contracts/ – This directory is for saving all the smart contracts.
- migrations/ – This directory contains scripts used for the deployment of smart contracts.
- test/ – This directory hosts the testing code for the smart contracts.

Finally, we need a file called truffle-config.js, which contains the configuration data and should be in the project root directory.

Change to the contracts directory using the following command:

```
cd contracts
```

Create a file named Coin.sol and copy the contract code from Listing 6-1 into the file, and then save the Coin.sol file within the contracts directory.

6.2.1 Contract Code

Listing 6-1. Code for ERC-20–based smart contract

```
pragma solidity 0.8.15;
interface ERC20Interface {
    function totalSupply() external view returns (uint);
    function balanceOf(address tokenOwner) external view
    returns (uint balance);
    function allowance(address tokenOwner, address spender)
    external view returns (uint remaining);
    function transfer(address to, uint tokens) external returns
    (bool success);
    function approve(address spender, uint tokens) external
    returns (bool success);
    function transferFrom(address from, address to, uint
    tokens) external returns (bool success);

    event Transfer(address indexed from, address indexed to,
    uint tokens);
    event Approval(address indexed tokenOwner, address indexed
    spender, uint tokens);
}

contract Coin is ERC20Interface {
    string public tokenSymbol;
    string public  tokenName;
    uint8 public tokenDecimals;
    uint public _totalSupplyOfToken;

    mapping(address => uint) tokenBalances;
    mapping(address => mapping(address => uint)) allowed;
```

```
//this is where we initialize our token with total supply ,
name etc
    constructor() public {
        tokenSymbol = "ISH";
        tokenName = "Isha Jain Coin";
        tokenDecimals = 2;
        _totalSupplyOfToken = 500000;
        tokenBalances[msg.sender] = _totalSupplyOfToken;
        emit Transfer(address(0), msg.sender, _total
        SupplyOfToken);
    }

// function to return the supply at any point in time.
    function totalSupply() public override view returns
    (uint) {
        return _totalSupplyOfToken  - tokenBalances
        [address(0)];
    }
 //function to check balance of tokens at a specific address
    function balanceOf(address tokenOwner) public override view
    returns (uint balance) {
        return tokenBalances[tokenOwner];
    }
 // function for transferring tokens to a specific address.
    function transfer(address to, uint tokens) public override
    returns (bool success) {
//checks if there is enough balance in the sender address
        require(tokens <= tokenBalances[msg.sender]);
//deduct tokens from the sender
        tokenBalances[msg.sender] = tokenBalances[msg.
        sender]-tokens;
        // add tokens to the recipient address
```

```
        tokenBalances[to] = tokenBalances[to] + tokens;
// once transfer is done emit a message
        emit Transfer(msg.sender, to, tokens);
        return true;
    }

    function approve(address spender, uint tokens) public
    override returns (bool success) {
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }
 // this is same as approve but here we can specify from
address which can be different from //message sender
        function transferFrom(address from, address to, uint
        tokens) public override returns (bool success) {

require(tokens <= tokenBalances[from]);
        tokenBalances[from] = tokenBalances[from]-tokens;

        require(tokens <= allowed[from][msg.sender]);
        allowed[from][msg.sender] = allowed[from][msg.
        sender]-tokens;

        uint c=0;
        c = tokenBalances[to] + tokens;
        require(c >=tokenBalances[to] );

        emit Transfer(from, to, tokens);
        return true;
    }
 // checks if tokenOwner is allowed to make the transfer
    function allowance(address tokenOwner, address
    spender)  public override view returns (uint remaining) {
```

```
        return allowed[tokenOwner][spender];
    }

    fallback() external payable {
        revert();
    }
}
```

Create a file called 2_deploy_contract.js in the migrations directory.

```
Copy the content

const Coin_Contract = artifacts.require("Coin");

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

Create a .env file in the root directory (mycoin directory) of the project with the following content:

```
INFURA_API_URL = "wss://ropsten.infura.io/ws/v3/<<your infura
project id>> "
MNEMONIC = "your meta mask mnemonic"

The INFURA_API_URL should use the wss (web socket) based
endpoint rather the https
```

Once you log in to your Infura account, you should see something similar to Figure 6-2.

ENDPOINTS ROPSTEN

https://ropsten.infura.io/v3/b1c76cbfffd24d09aa70726f91de1004

wss://ropsten.infura.io/ws/v3/b1c76cbfffd24d09aa70726f91de1004

Figure 6-2. *Different endpoints for Ropsten network*

We can obtain the mnemonic from our MetaMask application by following the screenshots shown in Figure 6-3 through Figure 6-5.

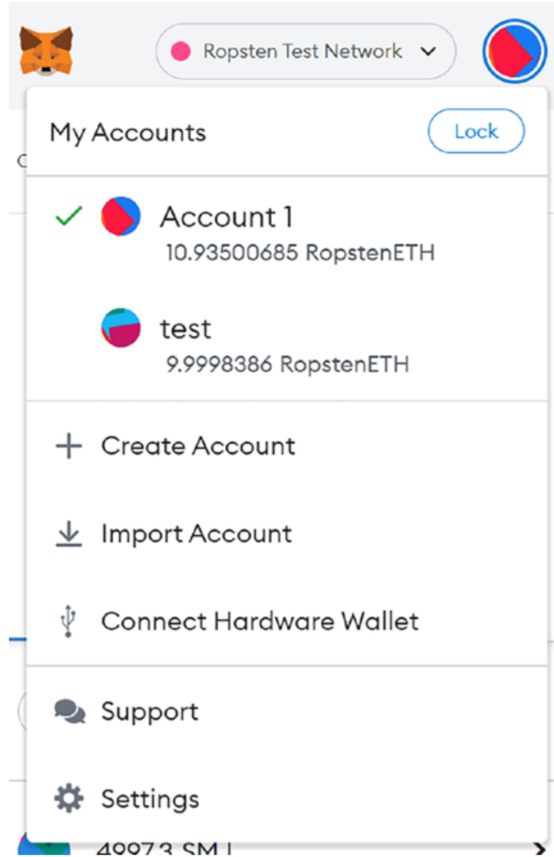

Figure 6-3. *MetaMask accounts*

Click on Settings, and you should see something similar to Figure 6-4.

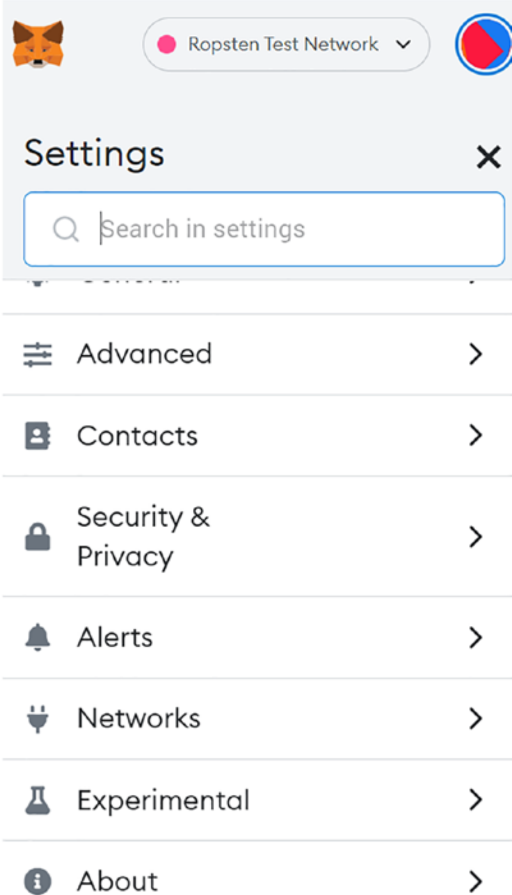

Figure 6-4. *Settings of the MetaMask*

Click Security & Privacy, and you should see the screen shown in Figure 6-5.

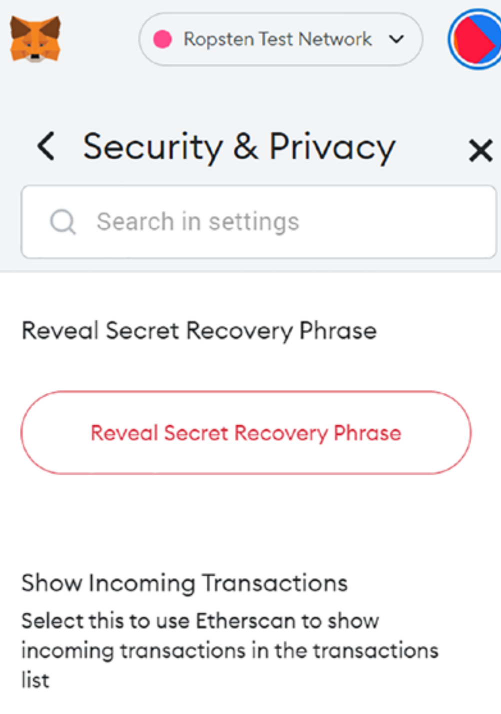

Figure 6-5. *Security and privacy settings in MetaMask*

Click on the Reveal Secret Recovery Phase button. It will ask for your MetaMask password and then will reveal your mnemonic.

There is a truffle-config.js file located within the project root directory (mycoin directory for the project we created). Modify the truffle-config.js file to include this content at the top:

```
require('dotenv').config();
const HDWalletProvider = require('@truffle/hdwallet-provider');
// this provides an indirection to load the Infura URL and
Mnemonic from the .env file
const { INFURA_API_URL, MNEMONIC } = process.env;
```

Also, in the Networks section of the truffle-config.js file, add the following content:

```
ropsten: {
      provider: () => new HDWalletProvider(MNEMONIC, INFURA_
      API_URL),
      network_id: 3,
      gas: 5500000,
      timeoutBlocks: 200,
      skipDryRun: true
    },
```

As we can see now, the Infura URL and mnemonic are loaded from the .env file. This indirection is needed for security purposes. The Ropsten network provider then uses the URL to connect to the Ropsten network and uses the mnemonic to interface with the MetaMask wallet.

6.2.2 Compile and Deploy the Contract

Compile the contract using the following command:

```
truffle compile
```

Deploy the contract using the following command:

```
truffle migrate --network ropsten
```

Once we run the command for deployment, we will see the result shown in Figure 6-6.

```
Starting migrations...
======================
> Network name:    'ropsten'
> Network id:      3
> Block gas limit: 30000000 (0x1c9c380)


3_deploy_contract.js
====================

   Deploying 'Coin'
   ----------------
   > transaction hash:    0x2eee8aa7e8ea0daa71ac3f7ed0b9af3f86bd6030098e7f15247671a75ed77bb0
   > Blocks: 1            Seconds: 8
   > contract address:    0x980991118BccbD7105eBb2cDD627c7059Fe1f278
   > block number:        12556255
   > block timestamp:     1657351584
   > account:             0xb96aeD3A4e11bBB1C028Ac96420305c803880Cd3
   > balance:             10.982393727700994873
   > gas used:            1039811 (0xfddc3)
   > gas price:           1.491261109 gwei
   > value sent:          0 ETH
   > total cost:          0.001550629705010399 ETH

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:     0.001550629705010399 ETH

Summary
=======
> Total deployments:   1
> Final cost:          0.001550629705010399 ETH
```

Figure 6-6. *Deployment of the ERC-20 coin via the Truffle interface*

We can go to Etherscan for the transaction 0x2eee8aa7e8ea0daa71ac3f7ed0b9af3f86bd6030098e7f15247671a75ed77bb0 and check its status, as shown in Figure 6-7. Go here:

`https://ropsten.etherscan.io/tx/0x2eee8aa7e8ea0daa71ac3f7ed0b9af3f86bd6030098e7f15247671a75ed77bb0`

Transaction Hash:	0x2eee8aa7e8ea0daa71ac3f7ed0b9af3f86bd6030098e7f15247671a75ed77t
Status:	Success
Block:	12556255 58 Block Confirmations
Timestamp:	13 mins ago (Jul-09-2022 07:26:24 AM +UTC)
From:	0xb96aed3a4e11bbb1c028ac96420305c803880cd3
Interacted With (To):	[Contract 0x980991118bccbd7105ebb2cdd627c7059fe1f278 Created]
Tokens Transferred:	From 0x0000000000000... To 0xb96aed3a4e11b... For 5,000 Isha Ja
Value:	0 Ether ($0.00)
Transaction Fee:	0.001550629705010399 Ether ($0.00)

Figure 6-7. *Etherscan view of the smart contract deployment*

We can also check the contract address at Etherscan. The contract address is 0x980991118BccbD7105eBb2cDD627c7059Fe1f278. Go here:

```
https://ropsten.etherscan.io/address/0x980991118BccbD7105eBb2c
DD627c7059Fe1f278
```

We just deployed a contract that creates a coin called ISH. The contract is deployed to the Ropsten testnet.

Now we will show how to interface with this contract using the Truffle console.

Open another command window. Navigate to the project directory (mycoin). Type in the following command:

```
truffle console -network ropsten
```

Once the console opens, issue the following command:

```
let instance = await Coin.deployed()
```

Once the smart contract is deployed on the Ropsten testnet, we can invoke the different functions on the contract via the Truffle framework itself.

There is a function called `balanceOf` that shows the balance of ISH tokens for the address.

Let's invoke it by using the following command:

```
instance.balanceOf("0xb96aeD3A4e11bBB1C028Ac9642030
5c803880Cd3")
```

In my case, the address I used is of the creator of the contract and as per the code of the contract this address owns all the tokens. On running the preceding command we will see the output as follows:

```
BN {
  negative: 0,
  words: [ 499000, <1 empty item> ],
  length: 1,
  red: null
}
```

We can see that there are 499,000 ISH tokens available with this address. Since I have already transferred some tokens to another address, we have less than 500,000 tokens.

I will inspect balance of the address to which I transferred ISH tokens by running the following command:

```
instance.balanceOf("0x1A703B299d764B4e28Dc2C7849CFe
DF9979D2430")
```

This shows the following output:

```
BN {
  negative: 0,
  words: [ 1000, <1 empty item> ],
```

```
  length: 1,
  red: null
}
```

Now, let's invoke the transfer function to transfer some ISH tokens.

Execute the following command in the console:

```
instance.transfer("0x1A703B299d764B4e28Dc2C7849CFeDF
9979D2430",1000)
```

Here, we are transferring 1000 ISH tokens to the address 0x1A703 B299d764B4e28Dc2C7849CFeDF9979D2430.

Upon execution of the command, we see the following output:

```
{
  tx: '0xd174faeb28e4b9c1d409c2cf10b7d215e52b44973e4bfc
  1350fb37a01282fb7a',
  receipt: {
    blockHash: '0x516715ace04223d3a36331140d7c569c62a2b4016
    a52b8d9b49d740efb41448f',
    blockNumber: 12556678,
    contractAddress: null,
    cumulativeGasUsed: 1294778,
    effectiveGasPrice: 1491261109,
    from: '0xb96aed3a4e11bbb1c028ac96420305c803880cd3',
    gasUsed: 35415,
    logs: [ [Object] ],
    logsBloom: '0x00000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000800000
    00000000000000000000000000000000000000000000810000000000
    000008000000000000000000000000000100004000000000000000000
    000000000000000000080000000000000000200000000000000000000
```

```
    0000000000000000010000000000000000000000000000000000000000000000
    0000000200000000000000000000000000000000000000000000008000000000 00
    00000000000000000000000000000000000000000000000000000000000000
    00001000000000',
    status: true,
    to: '0x980991118bccbd7105ebb2cdd627c7059fe1f278',
    transactionHash: '0xd174faeb28e4b9c1d409c2cf10b7d215e
    52b44973e4bfc1350fb37a01282fb7a',
    transactionIndex: 19,
    type: '0x0',
    rawLogs: [ [Object] ]
  },
  logs: [
    {
      address: '0x980991118BccbD7105eBb2cDD627c7059Fe1f278',
      blockHash: '0x516715ace04223d3a36331140d7c569c62a2b40
      16a52b8d9b49d740efb41448f',
      blockNumber: 12556678,
      logIndex: 18,
      removed: false,
      transactionHash: '0xd174faeb28e4b9c1d409c2cf10b7d215e52b4
      4973e4bfc1350fb37a01282fb7a',
      transactionIndex: 19,
      id: 'log_1e6ce129',
      event: 'Transfer',
      args: [Result]
    }
  ]
}
```

We can validate the transaction on Etherscan for the transaction ID 0xd174faeb28e4b9c1d409c2cf10b7d215e52b44973e4bfc1350fb37a01282fb7, as shown in Figure 6-8, by going to the following:

```
https://ropsten.etherscan.io/tx/0xd174faeb28e4b9c1d409c2cf10
b7d215e52b44973e4bfc1350fb37a01282fb7a
```

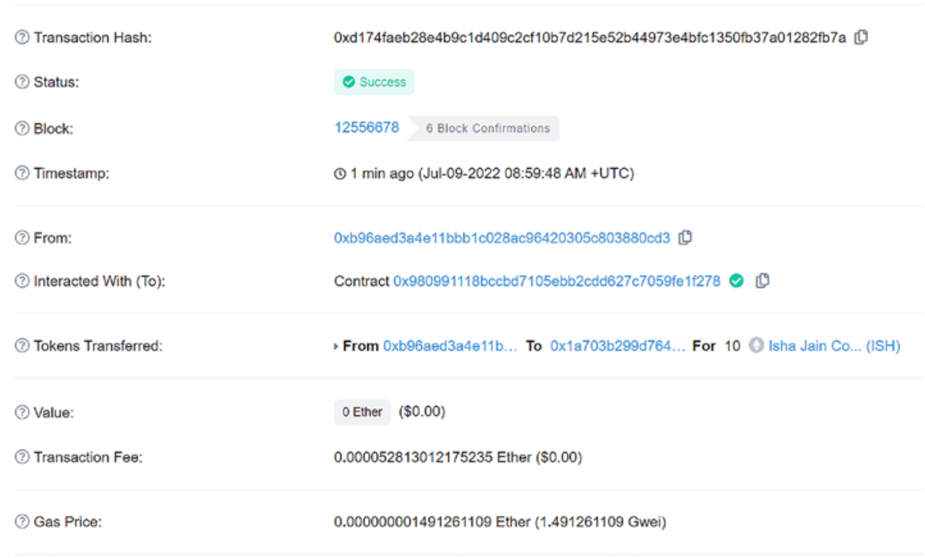

Figure 6-8. *Etherscan view of the transaction for checking the balance*

We can see it says 10 ISH coins, as we chose to go to two decimal places. So 1,000 tokens get divided by 100.

Now, let's check the balance of both the transferor and transferee.

Run the following command in the console to check the balance of the transferor:

```
instance.balanceOf("0xb96aeD3A4e11bBB1C028Ac9642030
5c803880Cd3")
```

We get the following output:

```
BN {
  negative: 0,
  words: [ 498000, <1 empty item> ],
  length: 1,
  red: null
}
```

Now, check the balance of the transferee by executing the following command:

```
instance.balanceOf("0x1A703B299d764B4e28Dc2C7849CFe
DF9979D2430")
```

This results in the following output:

```
BN {
  negative: 0,
  words: [ 2000, <1 empty item> ],
  length: 1,
  red: null
}
```

We can see that 1,000 ISH tokens got deducted from the transferor and added to the transferee.

6.3 Summary

In this chapter, we looked into one of the very famous frameworks for smart contract development, known as Truffle. We used the Truffle framework to create an ERC-20–based token and deployed it to the Ropsten test network. We also learned how to interact with that token by invoking different token functions using the Truffle framework.

In the next chapter, we will take a bit of a detour to learn about an alternate decentralized storage technology called Inter Planetary File System (IPFS). We will store digital assets in IPFS and then see how we can use the ERC-751 standard to create an NFT using the Remix web IDE.

CHAPTER 7

IPFS and NFTs

When you think of creating a decentralized application, you definitely have a blockchain platform like Ethereum in mind. The use of blockchain technology is extremely beneficial for the management of states, the automation of operations through the use of smart contracts, and the trading of economic value.

But where exactly does the content of your application get stored? Images? Videos? Where exactly are all of the HTML, CSS, and JavaScript files that make up the application's front end? Is the content that your users access as well as the application that you use loaded from a centralized AWS server?

The content would have to be stored on the blockchain, which is a time- and money-consuming process. Your blockchain application needs storage that is decentralized!

7.1 IPFS

The Inter Planetary File System (IPFS) is a peer-to-peer hypermedia protocol that aims to make the World Wide Web more open, quicker, and safer.

IPFS is a protocol for the storage and distribution of content. Every user acts as a node, just like in the realm of blockchain technology (server). The nodes are able to communicate with one another and share files with one another.

© Shashank Mohan Jain 2023
S. M. Jain, *A Brief Introduction to Web3*, https://doi.org/10.1007/978-1-4842-8975-4_7

To begin, the IPFS is considered decentralized due to the fact that the content is loaded from a network of thousands of peers rather than a single centralized server. Every single bit of data is hashed using cryptography, which produces a secure and one-of-a-kind type of content identification known as CID.

Since peer-to-peer file sharing has been available for some time, one might wonder what makes IPFS so unique in this field. IPFS is now the most well-known and widely used alternative for a decentralized internet because it possesses a number of advantageous characteristics that set it apart in the cryptocurrency industry.

IPFS is immutable, which means that once data has been added to the network, it cannot be altered in any way. Customers are able to check that the data they viewed has not been changed in any way. It is possible to publish updates, but these will always appear in the form of new files; the existing ones will never be overwritten.

IPFS prevents data from being duplicated by first chunking it, then storing it, and finally hashing it when it is added to the network. This allows for duplicate data to map to the same nodes, and as a result only one entry is made. Assuming the new file is somewhat comparable to those already present in the network, adding it to the network will require a smaller amount of storage space as the network grows larger.

IPFS is decentralized, which means that the network can continue to function even if some nodes are removed or added. Even if a significant number of nodes were taken offline, the system as a whole would continue to function normally. It would not be possible to delete information or censor files without first destroying each and every node that made up the network.

The decentralization of this system is the most important aspect. There are some representations coming up that compare and contrast the appearance of centralized networks with that of dispersed networks. The crucial point here is that Facebook effectively owns the first network,

and as such they are the only trustworthy source of information on the network. On the second network, there is no one node that is more significant than any of the others.

7.1.1 IPFS: 30,000-Foot View

Before we get started, let's take a high-level look at what IPFS accomplishes behind the scenes so we know what we're getting into. As a node in the IPFS network, you are responsible for establishing connections to dozens of other nodes located around the network and serving as a routing point for those nodes. You refer to these people as your peers.

Every node in the network is either a client or a piece of content at some point. You, as a customer, are responsible for keeping a list of peers organized to form the list you can have. Having a few acquaintances on the opposite side of the network (those whose IDs are substantially different from your own) and a large number of friends in the immediate area (whose IDs are very similar).

When requesting content, either because you desire it or because a peer asked for it, you have the option of limiting your search to those peers that you are familiar with that are located closest to the content ID. The request will be completed in an average of Log(N) amount of time. In addition, while the content is being returned, each node creates a cache of the data so that it can be accessed more quickly during subsequent searches. These caches are extremely aggressive and cut down network costs as a result of the fact that data can never change.

After that, you may either utilize a node that is running locally or route through an existing node by giving it the content ID. Either way, you will be able to retrieve the material that was stored.

7.1.2 Installation

For Windows, download the IPFS daemon from https://dist.ipfs.io/#go-ipfs.

Another option is to install the IPFS desktop application from https://docs.ipfs.tech/install/ipfs-desktop/.

In this chapter, we will work with the IPFS daemon:

```
C:\Users\I074560>cd \shashank\apress\web3
C:\shashank\apress\web3>mkdir ipfs
C:\shashank\apress\web3>cd kubo
```

```
C:\shashank\apress\web3\kubo>ipfs.exe init
```

This gives the following output:

```
generating ED25519 keypair...done
peer identity:
12D3KooWRzcEvQWq8wvr4AckSVVjRtUAvXLxc37J8uVLaGa2Bt4G
initializing IPFS node at c:\shashank\apress\web3\ipfs
```

To get started, enter the following:

```
        ipfs cat /ipfs/
QmQPeNsJPyVWPFDVHb77w8G42Fvo15z4bG2X8D2GhfbSXc/readme
```

Then, run the following command:

```
ipfs cat /ipfs/
QmQPeNsJPyVWPFDVHb77w8G42Fvo15z4bG2X8D2GhfbSXc/readme
```

This will check the IPFS installation, as shown in Figure 7-1.

Figure 7-1. *IPFS console*

Now let's add content to IPFS. Open another terminal and run the following command:

```
echo "This is my first content to IPFS" | ipfs add
```

This returns the following output:

```
added QmdWENEwy4RRdvfuR6Jk86AMe7TV89yUFZec2YKHZqAKqF
```

The hash returned is the content identifier (CID) for the content.

7.2 ERC-721

We saw in Chapters 5 and 6 how to use the ERC-20 standard for creating a token. But a token is a fungible entity. This means that each token is equivalent to another token. Many requirements for applications need different values for different assets, like the value of one piece of digital art cannot be equated to that of another piece of digital art. We need a mechanism to represent the value of such assets on the blockchain. This requirement gave birth to the ERC-721 specification, which deals with non-fungible tokens (NFTs, in short).

ERC-721 provides an interface that defines the functions for building non-fungible tokens. The following is a list of all of the functions and events that have been defined by the ERC-721 standard.

The ERC-721 standard defines a few functions that are compliant with the ERC-20 standard. Existing wallets will find it much simpler to display basic token information as a result of this change.

Functions Similar to ERC-20:

name: This field is where the name of the token is defined.

symbol: This field captures the symbol for the token.

totalSupply: This function represents the total supply of the NFT. The supply can be dynamic as well.

balanceOf: This function returns the total number of NFTs that are owned by an address.

Main Functions Related to Token Ownership:

ownerOf: This function gives back the address of the person who currently possesses a token. Due to the fact that ERC-721 tokens are non-fungible and individually identifiable, each token has its own ID that is stored on the blockchain.

approve: This function allows the owner of the NFTs to grant permissions for another account to transfer tokens on their behalf.

transfer: This function allows the owner of the NFT to transfer it to another address.

Metadata-related Function:

tokenMetadata: As the name suggests, this function allows the discovery of the metadata of the NFT; for example, what kind of data this NFT is holding, whether it's an audio or an image or something else.

Apart from these functions, there are two events defined as per the ERC-721 spec.

Transfer: When the token's ownership is transferred from one person to another, this event is triggered so that subsequent users can take control of the token.

Approve: This event is triggered whenever the `approve` function is called, which means that it occurs whenever a user gives permission to another user for that user to take ownership of a token.

7.3 Creating an ERC-721 Token and Deploying It to IPFS

With a high-level introduction to IPFS as well as NFTs, it's time to take a look at a simple use case where we use both these technologies.

Our intended use case is to create a digital asset and upload it to IPFS.

Before adding the digital asset to IPFS, make sure you start the IPFS daemon using the following command:

```
Ipfs daemon
```

The IPFS daemon will start as shown in Figure 7-2.

```
> Initializing daemon...
> API server listening on /ip4/127.0.0.1/tcp/5001
> Gateway server listening on /ip4/127.0.0.1/tcp/8080
```

Figure 7-2. *IPFS daemon*

Once the daemon is started, we will add an image to the IPFS using the following command by running the command from a different console:

```
Ipfs add <<filename>>
```

This will give us a hash (CID) as a result. Once we get this CID, to percolate the file to other nodes we just use the following `pin` command:

```
Ipfs pin add <<cid obtained above>>
```

This will make the content available via IPFS commands like the following:

```
Ipfs object get <<CID>>
```

This will return the image uploaded to IPFS.

We can also use a gateway like ipfs.io or cloudflare to get the content.

For this example, the asset is available at the following:

```
https://cloudflare-ipfs.com/ipfs/
QmbBp5huHazG582ean3eGGwWXkkVKnRc7V5te16ByWNs2N
https://ipfs.io/ipfs/
QmbBp5huHazG582ean3eGGwWXkkVKnRc7V5te16ByWNs2N
```

Here `QmbBp5huHazG582ean3eGGwWXkkVKnRc7V5te16ByWNs2N` is the CID of the content (image).

Once we have uploaded the image, we will create a json file that captures the metadata of the asset. The image URL is the IPFS URL via the ipfs.io gateway.

```
{
    "name": "Shashank Jain NFT",
    "description": "This image shows image of a laptop",
    "image": "https://ipfs.io/ipfs/
QmbBp5huHazG582ean3eGGwWXkkVKnRc7V5te16ByWNs2N",
}
```

Name the file as nft.json.

Add it to IPFS by using the following command:

```
Ipfs add nft.json
```

Pin it by using the following command:

```
Ipfs pin add <<hash obtained from ipfs add command output>>
```

In my case I have this json available at

```
https://cloudflare-ipfs.com/ipfs/
QmZwaouD8bVMoNgznszxsSAS7W9EgkdfC5rF7TSzz1Q25N
```

and

```
https://ipfs.io/ipfs/
QmZwaouD8bVMoNgznszxsSAS7W9EgkdfC5rF7TSzz1Q25N
```

Now that this part is done, we will create the smart contract using the Remix IDE.

Open Remix IDE in the browser by navigating to `https://remix.ethereum.org/`.

Create a file under the Contracts directory by the name of MyNFT.sol.

Copy the code from Listing 7-1 to the file.

Listing 7-1. ERC-721 implementation

```
//SDPX-License-Identifier: MIT
pragma solidity 0.8.0;

import "https://github.com/0xcert/ethereum-erc721/src/
contracts/tokens/nf-token-metadata.sol";
import "https://github.com/0xcert/ethereum-erc721/src/
contracts/ownership/ownable.sol";

contract MyNFT is NFTokenMetadata, Ownable {
```

```
  constructor() {
    //name of the NFT
    nftName = "Shashank Jain NFT";
   // nft symbol
    nftSymbol = "LAPTOP";
  }
 // this function provides the mint functionality with _to
variable will store the address of the
//receiver of the nft. tokenId stores the token identifier and
the uri stores the referenced url to the
// actual file location. In our case we have stored the file in
ipfs so the ipfs url will be used.
  function mint(address _to, uint256 _tokenId, string calldata
_uri) external onlyOwner {
    super._mint(_to, _tokenId);
    super._setTokenUri(_tokenId, _uri);
  }

}
```

Specifying the SPDX license type on Line 1 is a new feature that was introduced after Solidity version 0.8. Skipping the comment will not result in an error; instead, you will receive a warning.

The Solidity version is declared on Line 2.

The `0xcert/ethereum-erc721` contracts are imported on Lines 4 and 5.

This contract inherits most of the code from the ERC-721 spec.

We implement the mint method, which takes three inputs:

1. Address to which the NFT has to be transferred
2. Token ID (a random number)
3. URI (refers to the URL of the metadata file we uploaded to IPFS)

Compile the contract. Once the compilation is successful, we need to deploy the contract.

Set the environment field to Web3 Injected so that we can connect it to MetaMaskF for signing the transaction, as shown in Figure 7-3.

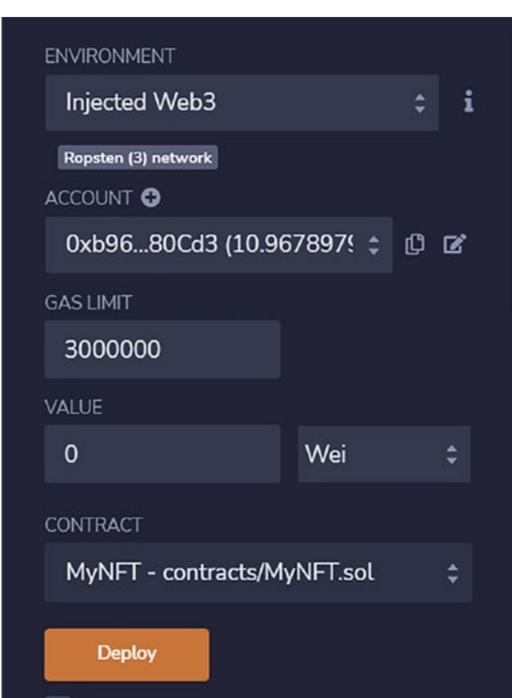

Figure 7-3. *Remix IDE for deploying the ERC-721 token*

Click on the Deploy button, and the contract should be deployed to the Ropsten test network, as shown in Figure 7-4.

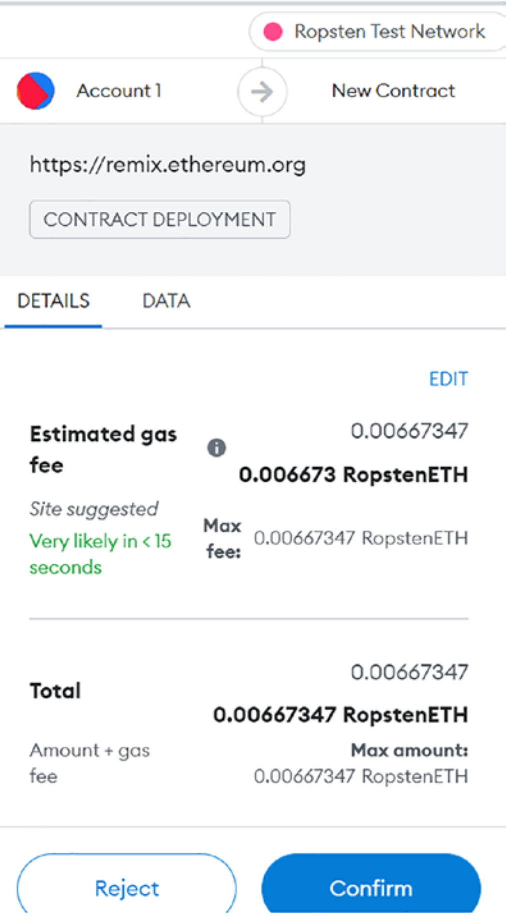

***Figure 7-4.** MetaMask wallet for approving the transaction*

Click the Confirm button. We can see the transaction is in a Pending state first, as shown in Figure 7-5.

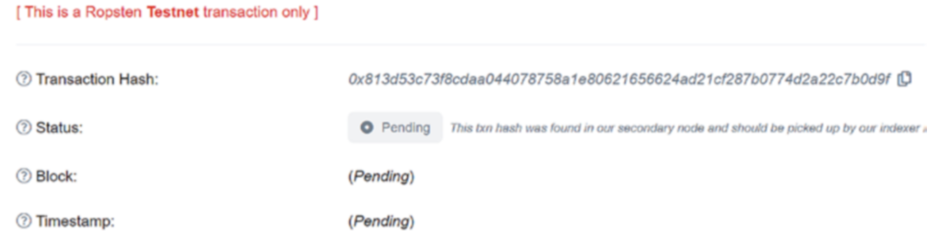

Figure 7-5. *Etherscan view of the ERC-721–based NFT deployment transaction*

It takes a few seconds before it is confirmed on the network.

We can see the transaction here on Etherscan:

```
https://ropsten.etherscan.io/tx/0x813d53c73f8cdaa044078758a
1e80621656624ad21cf287b0774d2a22c7b0d9f
```

Now we will invoke some of the functions of the contract via Remix. We can see the contract functions in Figure 7-6.

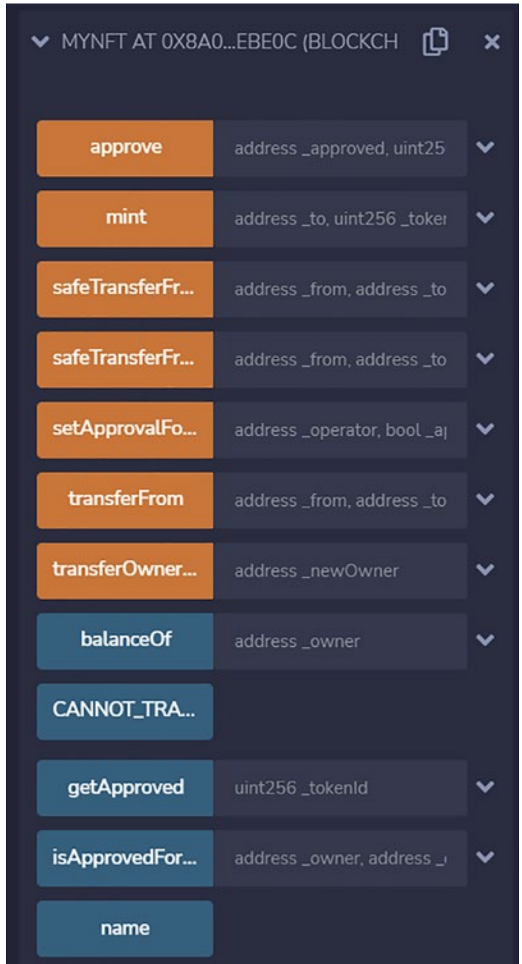

Figure 7-6. *ERC-721 functions in Remix IDE*

Some are inherited. We have overridden the mint function. Let's first mint an NFT, as shown in Figure 7-7.

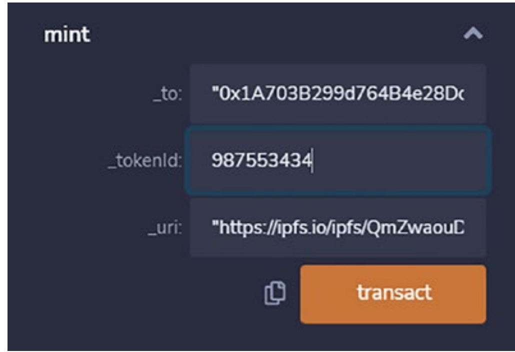

Figure 7-7. *Mint function in Remix IDE. We populate a random ID for the token ID and IPFS URL for URI and the receiver address in the _to field*

We have three inputs to the function as were discussed earlier.

We click on the Transact button now.

We can see the transaction on Etherscan after the confirmation via MetaMask, as shown in Figure 7-8.

[This is a Ropsten **Testnet** transaction only]

Transaction Hash: 0x2773297fcaf9ce4148fae84aafb5401d6041a57c319d682a5c789ff9067d3f23

Status: Pending *This txn hash was found in our secondary node and should be picked up by our indexer in*

Block: *(Pending)*

Timestamp: *(Pending)*

From: 0xb96aed3a4e11bbb1c028ac96420305c803880cd3

To: 0x8a062615bf8d8733f7eb86e09184e744561ebe0c

Figure 7-8. *Etherscan view of the NFT*

Once confirmed, we see the following on Etherscan, as shown in Figure 7-9.

Transaction Hash: 0x2773297fcaf9ce4148fae84aafb5401d6041a57c319d682a5c789ff9067d3f23
Status: Success
Block: 12570438 7 Block Confirmations
Timestamp: 1 min ago (Jul-11-2022 09:26:12 AM +UTC)
From: 0xb96aed3a4e11bbb1c028ac96420305c803880cd3
Interacted With (To): Contract 0x8a062615bf8d8733f7eb86e09184e744561ebe0c
Tokens Transferred: From 0x0000000000000... To 0x1a703b299d764...
For ERC-721 Token ID [987553434] Shashank Jai... (LAPTOP)

Figure 7-9. *Etherscan view of the NFT transaction*

We can see here that we have transferred the NFT to the 0x1A703B299d764B4e28Dc2C7849CFeDF9979D2430 address.

Token details can be found here, as shown in Figure 7-10.

Token Shashank Jain NFT

Overview

TokenID: 987553434

Transfers: 1

Transfers Contract

A total of 1 transaction found

	Txn Hash	Method
	0x2773297fcaf9ce4148f...	Mint

Figure 7-10. *Etherscan view of the NFT*

The Etherscan address for this is `https://ropsten.etherscan.io/token/0x8a062615bf8d8733f7eb86e09184e744561ebe0c?a=987553434`.

We initiate one more transfer of NFT to the same address, and then in Remix click on the `balanceOf` function by providing the address of the NFT beneficiary, as shown in Figure 7-11

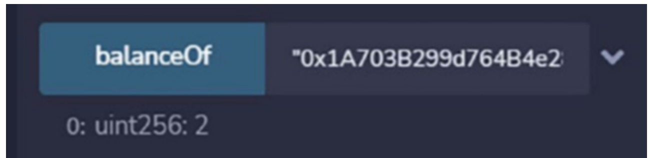

Figure 7-11. *balanceOf function shown in Remix IDE*

We see the address has received two NFTs into their account.

To get the owner of the contract, we click Owner. We get the result shown in Figure 7-12.

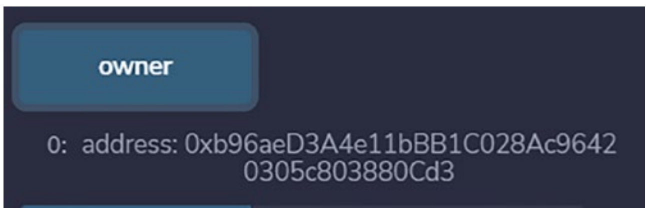

Figure 7-12. *Owner function shown in Remix IDE*

You can check on the owner of an NFT by passing the token ID, as shown in Figure 7-13.

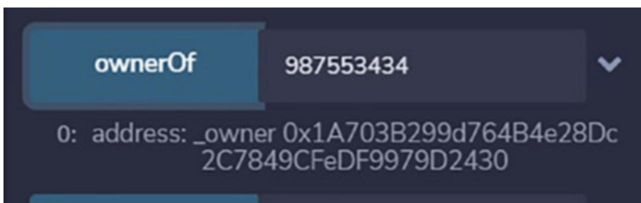

Figure 7-13. *OwnerOf function in Remix IDE*

Next, we check the `tokenURI` function, as shown in Figure 7-14.

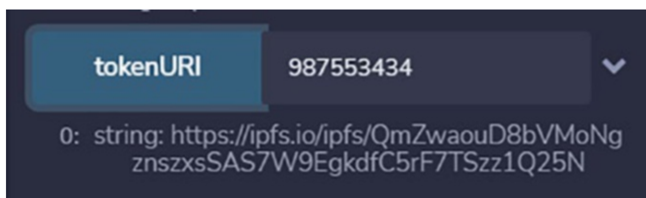

Figure 7-14. *Roken URI (IPFS URI in this case) as shown in Remix IDE*

Here again we pass the token ID as input. We can see it returns the URL of the metadata of the NFT stored in IPFS:

```
https://ipfs.io/ipfs/
QmZwaouD8bVMoNgznszxsSAS7W9EgkdfC5rF7TSzz1Q25N
```

Figure 7-15 shows the content of the file.

```
{
    "name": "Shashank Jain NFT",
    "description": "This image shows image of a laptop",
    "image": "https://ipfs.io/ipfs/QmbBp5huHazG582ean3eGGwWXkkVKnRc7V5te16ByWNs2N",
}
```

Figure 7-15. *Content of the file used for NFT*

We leave it to the reader now to try the same exercise using Truffle.

7.4 Summary

In this chapter, we looked at IPFS, which is the InterPlanetary File System. We looked at how we can upload and retrieve content via IPFS.

Next, we explored the ERC-721 spec, which is about the creation of NFTs. We created a simple NFT using Remix and IPFS.

In the next chapter, we will get a brief overview of another framework for the smart contract lifecycle, which is known as hardhat.

CHAPTER 8

Hardhat

In previous chapters, we have seen how to manage the lifecycle of Ethereum-based smart contracts using Remix IDE and Truffle. Now we will look at another framework known as Hardhat that can be used to manage the lifecycle of smart contracts. The lifecycle includes tasks like creating, compiling, testing, and deploying the contracts.

You can create smart contracts while having the assistance of Hardhat throughout the entire process. It is also of great assistance when testing already implemented contracts and developing "future assumptions."

8.1 Installation of Hardhat Framework

Install node.js (version 16.16.0) from the following link:

```
https://nodejs.org/en/download/
```

The following will include npm 8.11.0 with it:

```
Use npm to install hardhat
npm install -d hardhat
```

8.2 Workflow for Hardhat

At this stage, contracts are given their coding form and tested. Because you need to test each and every line of code, writing smart contracts and testing code typically go hand in hand. Because it offers some very

© Shashank Mohan Jain 2023
S. M. Jain, *A Brief Introduction to Web3*, https://doi.org/10.1007/978-1-4842-8975-4_8

nice plugins for testing and optimizing the code, Hardhat performs exceptionally well in this regard.

During the deployment step, you will first compile the code, which will involve converting the Solidity code into bytecode. Next, you will optimize the code before deploying it. There are a lot of great plugins available for Hardhat.

Let's begin a brand-new Hardhat project now that Hardhat has been successfully installed. We'll use npx to do so. Npx helps process node.js executables.

```
Create a project directory named coin
And cd into the coin directory
Use the command npx hardhat
```

```
C:\shashank\apress\web3>npx hardhat
npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.
Need to install the following packages:
  hardhat
Ok to proceed? (y) y
888    888                      888 888               888
888    888                      888 888               888
888    888                      888 888               888
8888888888  8888b.  888d888 .d88888 88888b.   8888b.  888888
888    888     "88b 888P"  d88" 888 888 "88b     "88b 888
888    888 .d888888 888    888  888 888  888 .d888888 888
888    888 888  888 888    Y88b 888 888  888 888  888 Y88b.
888    888 "Y888888 888     "Y88888 888  888 "Y888888  "Y888

Welcome to Hardhat v2.10.0

? What do you want to do? ...
  Create a JavaScript project
  Create a TypeScript project
> Create an empty hardhat.config.js
  Quit
```

Figure 8-1. *Hardhat installation*

Install the following dependencies:

```
npm install --save-dev @nomiclabs/hardhat-ethers ethers
@nomiclabs/hardhat-waffle ethereum-waffle chai
```

```
Cd to the coin directory
```

We will start by creating a contract.

Create a directory named Contracts inside the Coin directory and copy the file in Listing 8-1 into the contracts directory.

Listing 8-1. Code for coin contract

```
// SPDX-License-Identifier: LGPL-3.0-only
pragma solidity 0.8.15;
interface ERC20Interface {
    function totalSupply() external view returns (uint);
    function balanceOf(address tokenOwner) external view
    returns (uint balance);
    function allowance(address tokenOwner, address spender)
    external view returns (uint remaining);
    function transfer(address to, uint tokens) external returns
    (bool success);
    function approve(address spender, uint tokens) external
    returns (bool success);
    function transferFrom(address from, address to, uint
    tokens) external returns (bool success);

    event Transfer(address indexed from, address indexed to,
    uint tokens);
```

```
    event Approval(address indexed tokenOwner, address indexed
    spender, uint tokens);
}

contract Coin is ERC20Interface {
    string public tokenSymbol;
    string public  tokenName;
    uint8 public tokenDecimals;
    uint public _totalSupplyOfToken;

    mapping(address => uint) tokenBalances;
    mapping(address => mapping(address => uint)) allowed;
//this is where we initialize our token with total supply ,
name etc
    constructor() public {
        tokenSymbol = "ISH";
        tokenName = "Isha Jain Coin";
        tokenDecimals = 2;
        _totalSupplyOfToken = 500000;
        tokenBalances[msg.sender] = _totalSupplyOfToken;
        emit Transfer(address(0), msg.sender, _total
        SupplyOfToken);
    }

// function to return the supply at any point in time.
    function totalSupply() public override view returns
     (uint) {
        return _totalSupplyOfToken  - tokenBalances
        [address(0)];
    }
 //function to check balance of tokens at a specific address
    function balanceOf(address tokenOwner) public override view
    returns (uint balance) {
```

```
        return tokenBalances[tokenOwner];
    }
 // function for transferring tokens to a specific address.
    function transfer(address to, uint tokens) public override
    returns (bool success) {
//checks if there is enough balance in the sender address
        require(tokens <= tokenBalances[msg.sender]);
//deduct tokens from the sender
        tokenBalances[msg.sender] = tokenBalances[msg.sender]
        -tokens;
        // add tokens to the recipient address
        tokenBalances[to] = tokenBalances[to] + tokens;
// once transfer is done emit a message
        emit Transfer(msg.sender, to, tokens);
        return true;
    }

    function approve(address spender, uint tokens) public
    override returns (bool success) {
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }
 // this is same as approve but here we can specify from
address which can be different from //message sender
        function transferFrom(address from, address to, uint
        tokens) public override returns (bool success) {

require(tokens <= tokenBalances[from]);
        tokenBalances[from] = tokenBalances[from]-tokens;

        require(tokens <= allowed[from][msg.sender]);
```

```
        allowed[from][msg.sender] = allowed[from][msg.
        sender]-tokens;

        uint c=0;
        c = tokenBalances[to] + tokens;
        require(c >=tokenBalances[to] );

        emit Transfer(from, to, tokens);
        return true;
    }
 // checks if tokenOwner is allowed to make the transfer
    function allowance(address tokenOwner, address
    spender)  public override view returns (uint remaining) {
        return allowed[tokenOwner][spender];
    }

    fallback() external payable {
        revert();
    }
}
```

8.3 Deployment of the Smart Contract

Create a file named hardhat.config.js inside the Coin directory as follows:

```
hardhat.config.js
/** @type import('hardhat/config').HardhatUserConfig */
require("@nomiclabs/hardhat-ethers")
require("@nomiclabs/hardhat-waffle")
module.exports = {
  solidity: "0.8.9",
  networks: {
    ropsten: {
```

```
      url: "https://ropsten.infura.io/v3/<<your infura
      project id>>",
      accounts: {
        mnemonic: "<<your mnemonic>>",
        path: "m/44'/60'/0'/0",
        initialIndex: 0,
        count: 20,
        passphrase: "",
      },
    },
  },
};
```

Create a directory named Deployments inside the Coin directory. Create a file called deployToken.js and copy the code from Listing 8-2.

Listing 8-2. Code for deployment

```
async function main() {
const Coin = await ethers.getContractFactory("Coin");
const coin = await Coin.deploy();
await coin.deployed();

console.log("Coin deployed to:", coin.address);
}

main()
.then(() => process.exit(0))
.catch((error) => {
    console.error(error);
    process.exit(1);
});
```

Deploy the contract to the Ropsten network using the following command:

```
npx hardhat run deployments/deployToken.js --network Ropsten
```

We will see the output shown in Figure 8-2.

```
C:\shashank\apress\web3>npx hardhat run deployments/deployToken.js --network ropsten
npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.
Coin deployed to: 0xc136c20061344B0D6096adB6edd651aaCEFE0D7e
```

Figure 8-2. *Output of deployment of ERC-20–based coin using hardhat*

The contract can be seen at Etherscan at this URL:

`https://ropsten.etherscan.io/address/0xc136c20061344B0D6096adB6edd651aaCEFE0D7e`

You can check the transaction at the following: `https://ropsten.etherscan.io/tx/0x048c3b3f04519df9b316363e177e26d1716ba62460ee4764b6d0297c6b7ccd93`

We can see the ISH coin tokens have been transferred to the creator address, as shown in Figure 8-3.

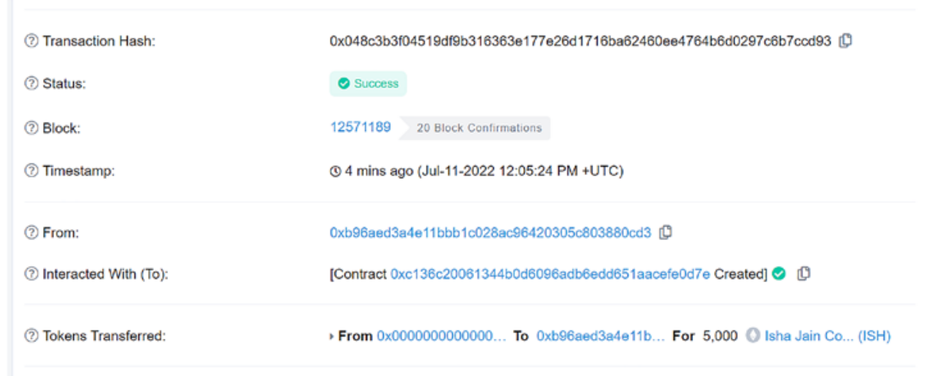

Figure 8-3. *Etherscan view of the deployment transaction*

To invoke functions on deployed contracts, create a file invoke.js inside the Deployments directory.

Listing 8-3. Code for invoking the smart contract function

```
async function main() {
const MyContract = await ethers.getContractFactory("Coin");
const contract = await MyContract.attach(
  "0xc136c20061344B0D6096adB6edd651aaCEFE0D7e" // The deployed
contract address
);

// Now you can call functions of the contract
console.log(await contract.balanceOf("0xb96aeD3A4e11bBB1C028
Ac96420305c803880Cd3"));
}
main()
.then(() => process.exit(0))
.catch((error) => {
    console.error(error);
    process.exit(1);
});
```

Invoke the function using the following mentioned command by running it from the Coin directory:

```
npx hardhat run deployments/invoke.js --network ropsten
```

We get the following output:

```
BigNumber { value: "500000" }
```

Create another file called transfer.js inside the Deployments directory and copy the code in Listing 8-4 into this file.

Listing 8-4. Code for Transferring coins

```
async function main() {
const MyContract = await ethers.getContractFactory("Coin");
const contract = await MyContract.attach(
  "0xc136c20061344B0D6096adB6edd651aaCEFE0D7e" // The deployed
contract address
);

// Now you can call functions of the contract
console.log(await contract.transfer("0x1A703B299d764B4e28Dc
2C7849CFeDF9979D2430",200));
}
main()
.then(() => process.exit(0))
.catch((error) => {
    console.error(error);
    process.exit(1);
});
```

Execute the code using the following command:

```
npx hardhat run deployments/transfer.js --network ropsten
```

We will see the output of the command, as shown in Figure 8-4.

```
npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.
{
  hash: '0xb86ebde83e168acd22a424f8d1255d317098a2bb57f92e6aa188f324bc2b28b6',
  type: 2,
  accessList: [],
  blockHash: null,
  blockNumber: null,
  transactionIndex: null,
  confirmations: 0,
  from: '0xb96aeD3A4e11bBB1C028Ac96420305c803880Cd3',
  gasPrice: BigNumber { value: "1500000008" },
  maxPriorityFeePerGas: BigNumber { value: "1500000000" },
  maxFeePerGas: BigNumber { value: "1500000008" },
  gasLimit: BigNumber { value: "52503" },
  to: '0xc136c20061344B0D6096adB6edd651aaCEFE0D7e',
  value: BigNumber { value: "0" },
  nonce: 44,
  data: '0xa9059cbb0000000000000000000000001a703b299d764b4e28dc2c7849cfedf9979d2430000000000000000000000000000000000
  r: '0xb3d18e28d25218b44c55342208d27746fbf3ecc62af160cdb759cae161d90037',
  s: '0x37d2d67d7f7af68d4b48580e239a53f11226c3fd02f76006f828d5bf7c430600',
  v: 0,
  creates: null,
  chainId: 3,
  wait: [Function (anonymous)]
}
```

Figure 8-4. *Hardhat output of the transfer function invocation of the smart contract*

We can see the transaction on Etherscan, as shown in Figure 8-5.

`https://ropsten.etherscan.io/tx/0xb86ebde83e168acd22a424f8d125 5d317098a2bb57f92e6aa188f324bc2b28b6`

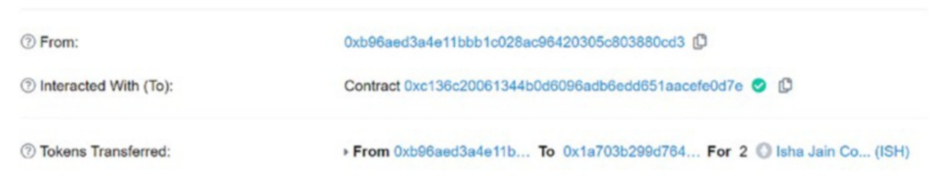

Figure 8-5. *Etherscan view of the smart contract transfer transaction*

We can see since the decimal place used is two, the transfer reflects two ISH coins.

We now check the balance of transferee address with the modified code of invoke.js (Listing 8-5).

Listing 8-5. Code for checking balance of an account

```
async function main() {
const MyContract = await ethers.getContractFactory("Coin");
const contract = await MyContract.attach(
  "0xc136c20061344B0D6096adB6edd651aaCEFE0D7e" // The deployed
contract address
);

// Now you can call functions of the contract, We call
balanceOf function by passing the transferee address.
console.log(await contract.balanceOf("0x1a703b299d764b4e28dc
2c7849cfedf9979d2430"));
}
main()
.then(() => process.exit(0))
.catch((error) => {
    console.error(error);
    process.exit(1);
});
```

Now execute the following command:

```
npx hardhat run deployments/invoke.js --network ropsten
```

This outputs the following:

```
BigNumber { value: "200" }
```

This matches with what we transferred to the transferee: 200 tokens. In MetaMask and Etherscan, this reflects as 2 ISH coins, since we used two decimals places as the configuration while creating the coin.

8.4 Summary

In this chapter, we looked into a framework called Hardhat for contract deployment and testing. We looked into the installation of Hardhat and then how we can use it for deployment of contracts. We also looked into how we can use the same framework for invoking the functions on the smart contract.

Index

A

Abstract contract, 52–54

B

Balance, 59–60, 125
Binance, 64, 66, 73
Binance Smart Chain (BSC), 73
Bitcoin, 6, 9, 12, 13, 15–23, 25, 26, 63, 109
Blockchain technology, 6, 11, 18, 147
- algorithm, 13
- applications, 16
- architecture
  - client–server network, 21
  - nodes, 22
  - transaction flow, 22
  - verification process, 22
- block, 14
- chain, 15
- classification, 11
- components, 14
- computers, 13
- cryptocurrency, 17
- data, 13
- database, 14
- data structure, 11
- network, 15
- permissioned, 12
- Postgres nodes, 11
- private, 12
- public, 12
- setup, 12

C

Capitalization, 33
Centralized authority, 3, 5, 13, 23
Coinbase, 32, 66
Coin.sol file, 129
Consensus, 17
- proof of work, 19

Content identifier (CID), 148, 151, 154
Contract codes, 26, 129–137
Cryptocurrencies, 14, 15, 17–19, 21, 25, 29, 64, 66, 109, 148
Cryptographic keys
- blockchain networks, 23
- data structure, 24
- genesis block, 24

Cryptography, 64, 148
Custodial wallets, 64, 65

© Shashank Mohan Jain 2023
S. M. Jain, *A Brief Introduction to Web3*, https://doi.org/10.1007/978-1-4842-8975-4

D

Decentralization, 6
  centralization, 4
  components, 4
  networks, 3
  setup, 3
Decentralized applications (DApps), 6, 25, 27–29, 66, 73, 109, 147
Decentralized system, 3–8
2_deploy_contract.js, 133
deployToken.js file, 173, 174
Desktop wallet, 65

E

ERC-20 standard, 109, 110, 152
  smart contract, 130
  functions, 110, 111
ERC-721 standard, 152, 153
  NFT deployment transaction, 159
  functions, 160
  implementation, 155
  token, 157
  Ether, 66
Ethereum, 6, 12, 13, 15, 17, 26, 27, 29, 61, 63, 109
  address, 35, 56
  APIs, 76
  blockchain, 20, 28, 63, 111
  ERC 20 standard–based tokens, 65
  wallets, 63
Ethereum virtual machine (EVM), 7, 28, 29, 60, 61
Etherscan, 70, 71, 75, 76, 104, 105, 107, 108, 117, 138, 139, 143, 159, 161–163, 174, 177, 178
Etherscan transaction view, 76
Externally owned accounts (EOA), 25, 26, 56, 60

F, G

Function modifier, 43–46

H

Hardhat, 167
  dependencies, 169
  deployment, ERC-20–based coin, 174
  deployment step, 168
  installation, 167
  smart contract
    deployment, 172–175
    function, 175
    transfer transaction, 177
  transaction on Etherscan, 177
  workflow, 167, 168
hardhat.config.js file, 172
Hardware wallet, 65
Hash functions, 19, 57
Hypertext markup language (HTML), 1, 82, 84, 147

I

Infura, 72, 73
- account creation page, 79, 80
- architecture, 73
- Create New Project screen, 80, 81
- default node provider, 74
- Ethereum API, 73, 74
- IaaS and Web3 backend provider, 72
- infrastructure, 73
- project ID and project secret, 81, 82
- project named trial, 81
- Ropsten network, 84
- Rropsten network, 82–87
- Web3 service, 72

Injected Web3 environment, 101
Integrated development environment (IDE), 62, 89, *See also* Remix IDE
Interfaces, 53–54, 65, 66, 72, 76, 110, 111, 137, 139, 152
Internet Protocol File System (IPFS), 147
- 256-bit hash, 149
- CID, 148
- content ID, 149
- decentralization, 148
- installation, 150, 151
- peer-to-peer file sharing, 148

InterPlanetary File System (IPFS), 72, 145, 147, 165
IPFS console, 151
ISH coin tokens, 174

J, K

JAIN, 114, 120, 122, 124, 125

L

Libraries, 54, 55
LinkedIn, 8
Loops, 37–39, 62

M

Merge wildcard, 44
MetaMask wallet, 66, 91, 134
- account details, 69, 70
- account on Etherscan, 70, 71
- confirmation of ETH, 75
- extension on Brave browser, 68
- HD wallet, 67
- Infura, 72–74
- installation and configuration instructions, 68
- networks available, 71, 72
- Ropsten ETH, 75
- Ropsten testnet, 74
- wallet extension, 69

Mobile wallet, 65
MyNFT.sol, 155

N, O

Networks, 4, 8
- cloud application, 5
- stand-alone application, 5

NFTokenMetadata, 155

Node Package Manager (NPM), 128
Non-fungible tokens (NFTs), 145, 147–165

P, Q

Peer-to-peer software, 5
Polygonnetwork, 73
Priority Gas Auction (PGA), 108
Private key, 23–25, 56–58, 63–65
Programming languages, 7, 17, 25, 28, 30, 34, 35, 37, 39, 41, 56, 76
Proof-of-stake algorithm, 20
Proof-of-stake blockchains, 20
Proof-of-work blockchains, 18

R

Remix IDE
  account address from MetaMask, 118
  balance in test account, 125
  browser-based environment, 89
  code for simple smart contract, 93
  compilation screen for smart contract, 96
  Compiled Test.sol, 96, 97
  compiler version and language, 97, 98
  contract functions, 105, 106
  for deployed contract, 119
  deployed contract transfer function, 120, 121
  deployment button, 98, 99
  deployment environments, 100, 101
  deployment screen, 115
  ERC-20 specs, 111
  ERC-721 functions, 160
  Etherscan view
    deployment transaction, 103–105
    smart contract function invocation transaction, 107, 108
    transaction just done, 117
  homepage, 92
  Icon Panel, 90
  Jain.sol, 111
  JAIN tokens, 124, 125
  Main Panel, 90
  message in the text box, 106
  MetaMask view
    account, 123
    transfer transaction, 122
  MetaMask wallet, 102–104
  panels, 89, 90
  Remix view, deployed contract, 118
  Ropsten test network for deployment, 101, 102
  setMessage function, 107
  Side Panel, 90
  smart contract creation, 91
  Solidity and MetaMask, 91
  Terminal, 90
  Test.sol, 92, 93

transaction approval in MetaMask, 115, 116
Ropsten network, 76, 82–87, 91, 101, 134, 137, 174
Ropsten test network, 74, 91, 102–104, 126, 127, 139, 140, 144, 157

S

Shashank Jain
Coin, 114
Smart contract, 7, 8, 17, 25–30, 39, 41, 42, 48, 56, 57, 59, 65, 66, 76, 77, 87, 89–91, 96, 101, 106–108, 110, 112, 127, 129–137, 139, 144, 147, 155, 165, 167, 172–178
Solidity, 26, 28, 30, 32, 33, 39, 41, 56
address data type, 59
arithmetic operations, 35
assignment, 37
comparison, 36
environment, 56
Ethereum, 57
keyword event, 55
fallback function, 48
function overloading, 50
functions, 41
logical, 36
pragma directive, 30
pure functions, 47
types, 34
variables, 31, 34
view functions, 46
visibility requirements, 51
Solidity programming language, 30

T

Testnets, 66
Goerli, 67
Kovan, 67
in operation, 67
Rinkeby, 67
Ropsten, 67
Test.sol, 92, 93, 95–97
Tokens, 66, 67, 109–126
Transfer function, 35, 60, 120, 121, 141, 177
transfer.js file, 175
Truffle, 127, 144
installation, 127, 128
Install node.js, 127
NPM, 128
smart contract deployment
Coin.sol, 129
compilation and deployment, contract, 137–144
contract code, 130, 133
MetaMask accounts, 134
mkdir mycoin, 129
security and privacy settings in MetaMask, 136
settings, MetaMask, 135
truffle init, 129

U

User interface (UI), 8

V

Validator nodes, 66

W

Wallets, 63
- as custodial wallets, 64
- desktop wallet, 65
- hardware wallet, 65
- MetaMask wallet, 66 (*see also* MetaMask wallet)
- mobile wallet, 65
- private and public keys, 64
- testnets, 67
- web extensions, 65

Web 1.0, 1
Web 2.0 applications, 2
Web 3.0, 2
- apps, 6, 8
- architecture, 7
- evolution, 2
- platforms, 8

web3.js library, 76
- web3-bzz module, 77, 78
- web3-eth module, 77
- web3-net module, 78
- web3-shh module, 77
- web3-utils module, 78

Web extensions, 65

X, Y, Z

0xcert/ethereum-erc721, 156