packt

2ND EDITION

Hands-On Blockchain for Python Developers

Empowering Python developers in the world of blockchain and smart contracts

ARJUNA SKY KOK

Hands-On Blockchain for Python Developers

Empowering Python developers in the world of blockchain and smart contracts

Arjuna Sky Kok

‹packt›

Hands-On Blockchain for Python Developers

Group Product Manager: Kunal Sawant
Publishing Product Manager: Akash Sharma
Senior Editor: Esha Banerjee
Technical Editor: Jubit Pincy
Copy Editor: Safis Editing
Project Coordinator: Manisha Singh
Indexer: Manju Arasan
Production Designer: Jyoti Kadam
Marketing Coordinator: Sonia Chauhan

First published: February 2019
Second edition: June 2024

Production reference: 1130624

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80512-136-7

www.packtpub.com

To the memory of my brother, Hengdra Santoso, for the unbreakable bond we shared as brothers.

– Arjuna Sky Kok

Contributors

About the author

Arjuna Sky Kok is a skilled software engineer with a passion for all things related to finance and technology. He lives in Jakarta, where he studied mathematics and programming at Binus University. Arjuna's academic achievements include double degrees in Computer Science and Mathematics. Currently, he is focusing his talent in the crypto space, as he believes that DeFi and NFT will serve as the foundation for future finance. He also has a keen interest in AI, especially Generative AI. Outside of work, Arjuna enjoys watching anime, listening to J-pop songs, and playing basketball.

About the reviewers

Ankit Anchila is a distinguished software development professional with over a decade of industry experience. With a background in software development, Ankit brings a wealth of expertise and insights to the table. Ankit has served as a judge at numerous hackathons and has evaluated projects for prestigious awards such as the Webby, Edison, and Globee Awards. Beyond his professional achievements, Ankit is an active contributor to the software development community. He regularly shares valuable insights through articles in tech blogs, influencing and inspiring fellow developers. As a member of *The International Academy of Digital Arts and Sciences* (IADAS), Ankit continues to shape the future of digital innovation.

Shivanjan Chakravorty is a distinguished Member of Technical Staff 3 at Pure Storage, where he brings a wealth of expertise to the forefront of technological innovation. Renowned for his contributions to open-source projects, especially libopenstorage, Shivanjan excels in Golang and Python. His expertise covers a wide range of technologies including Kubernetes, Machine Learning, and Generative AI. He has also worked extensively with Large Language Models (LLMs). Additionally, his scholarly work includes two published research papers in the domain of mathematical optimization. Beyond his professional pursuits, Shivanjan is an avid adventurer and a chess enthusiast, embodying a dynamic blend of intellect and passion.

Table of Contents

3

Using Vyper to Implement a Smart Contract 51

Part 2: Web3 and Ape Framework

4

Using Web3.py to Interact with Smart Contracts 77

5

Ape Framework 105

6

Building a Practical Decentralized Application 123

Part 3: Graphical User Interface Applications

7

Front-End Decentralized Application 145

8

Cryptocurrency Wallet 165

Part 4: Related Technologies

9

10

11

Part 5: Cryptocurrency and NFT

12

13

Part 6: Writing Complex Smart Contracts

14

Preface

The blockchain revolution is well underway, and its impact is being felt across various industries. From finance and media to governance, the decentralized and transparent nature of blockchain technology is driving innovation and disrupting traditional models. At the heart of this revolution lies the power of smart contracts – self-executing programs that facilitate secure and trustless transactions without the need for intermediaries.

As a Python developer, you are already equipped with a powerful and versatile programming language that has gained widespread adoption in various domains, including data science, web development, and automation. By combining your Python skills with the world of blockchain and smart contracts, you can unlock new opportunities and build revolutionary decentralized applications (dApps).

This book, *Hands-On Blockchain for Python Developers*, is designed to be your comprehensive guide to navigating the exciting realm of blockchain programming. With a strong emphasis on practical applications, you will embark on a journey that takes you from the fundamentals of blockchain and smart contract development to building real-world dApps using cutting-edge tools and frameworks.

Throughout the chapters, you will gain hands-on experience with Ethereum, learn to write smart contracts using languages such as Vyper, and explore the intricacies of interacting with these contracts using Python's `web3` library. You will also delve into advanced topics such as decentralized file storage with IPFS, second-layer (L2) solutions, token creation, non-fungible tokens (NFTs), and decentralized finance (DeFi) applications like Decentralized Exchange (DEX).

Python's familiarity and ease of use will accelerate your learning curve as you venture into the high-demand field of smart contract and blockchain development.

Who this book is for

Python developers, smart contract software engineers, and blockchain developers can gain practical insights into how to write smart contracts and build blockchain applications.

The three main personas who are the target audience of this content are as follows:

- Python developers: Developers who know Python already and want to build blockchain applications.
- Smart contract software engineers: They will learn to use Vyper to write smart contracts. This book will help them in their career in writing complex smart contracts.

- Blockchain developers: They will gain an overview of how to integrate smart contracts with non-blockchain applications. This will help them succeed in interacting with smart contracts from web applications or desktop applications.

What this book covers

Chapter 1, *Introduction to Blockchain Programming*, provides an overview of the fundamental concepts and principles underlying blockchain technology. It starts by exploring the traditional centralized system's limitations and how blockchain offers a decentralized, transparent, and immutable solution.

Chapter 2, *Smart Contract Fundamentals*, provides an overview of core concepts and mechanics behind smart contracts. It begins by defining what smart contracts are and their key characteristics, such as deterministic execution, immutability, and trustless interactions.

Chapter 3, *Using Vyper to Implement a Smart Contract*, provides an overview of the Vyper language so readers will be able to write and deploy smart contracts on the Ethereum blockchain. Readers will learn the syntax, data types, and control structures of Vyper.

Chapter 4, *Using Web3.py to Interact with Smart Contracts*, explores how to interact with deployed smart contracts using the `Web3.py` library. Readers will learn how to connect to Ethereum nodes, load contract ABIs, and invoke contract functions, enabling them to build decentralized applications that seamlessly interact with smart contracts.

Chapter 5, *Ape Framework*, introduces readers to the Ape development framework, a tool that streamlines the process of building, testing, and deploying smart contracts and decentralized applications. Readers will learn how to leverage Ape's features, such as contract compilation, unit testing, and deployment scripts to efficiently manage the development lifecycle of their blockchain projects.

Chapter 6, *Building a Practical Decentralized Application*, puts the knowledge gained so far into practice by guiding readers through the development of a decentralized voting application. Leveraging the skills acquired in previous chapters, readers will implement smart contracts for secure and transparent voting.

Chapter 7, *Front-End Decentralized Application*, focuses on developing a user-friendly graphical interface for the decentralized voting application built in the previous chapter, utilizing the `PySide` library. Readers will learn how to integrate their smart contracts with a modern GUI, enabling seamless interaction with the blockchain-based voting system and providing an intuitive experience for end-users.

Chapter 8, *Cryptocurrency Wallet*, guides readers through the process of building a user-friendly cryptocurrency wallet application using the `PySide` library. The wallet will be able to transfer ETH to a user and save the user's address into an address book.

Chapter 9, *InterPlanetary: A Brave New File System*, introduces readers to the InterPlanetary File System (IPFS), a revolutionary decentralized storage system that enables secure and resilient file sharing across a peer-to-peer network. Readers will learn how IPFS addresses the limitations of traditional centralized storage solutions.

Chapter 10, *Implementing a Decentralized Application Using IPFS*, dives into the practical implementation of a decentralized video-sharing application that leverages the power of IPFS. Readers will learn how to install and interact with the `IPFS` library, write a smart contract for managing video paths, and develop a user-friendly web application that enables decentralized video sharing across the IPFS network.

Chapter 11, *Exploring Layer 2*, introduces readers to the concept of Layer 2 (L2) solutions. It aims to address the scalability and throughput challenges of the Ethereum mainnet. In this chapter, readers will learn about various L2 technologies and gain hands-on experience deploying a smart contract onto one of these L2 platforms, enabling them to build high-performance and cost-effective decentralized applications.

Chapter 12, *Creating Tokens on Ethereum*, delves into the process of creating custom tokens on the Ethereum blockchain. Readers will learn how to develop a simple token smart contract, understand the widely adopted ERC-20 token standard, and explore mechanisms for selling and distributing tokens, enabling them to participate in the world of tokenized assets and decentralized finance.

Chapter 13, *How to Create an NFT*, introduces readers to the concept of non-fungible tokens (NFTs) and their unique properties. Readers will explore the ERC-721 standard, which governs the creation and management of NFTs on the Ethereum blockchain, and gain hands-on experience in developing a smart contract that mints and transfers ownership of these unique digital assets.

Chapter 14, *Writing NFT Marketplace Smart Contracts*, dives deep into the world of NFT marketplaces, enabling readers to create platforms for buying, selling, and trading non-fungible tokens. Readers will learn how to design and implement an NFT marketplace smart contract, write comprehensive tests to ensure its correctness, and explore advanced features to enhance the user experience of the marketplace.

Chapter 15, *Writing a Lending Vault Smart Contract*, explores the concept of lending vaults in the decentralized finance (DeFi) space. Readers will gain insights into the ERC-4626 standard, which governs the implementation of these vaults, and learn how to develop a smart contract that facilitates secure and transparent lending operations on the Ethereum blockchain.

Chapter 16, *Decentralized Exchange*, takes readers on a journey through the world of decentralized finance (DeFi) by exploring decentralized exchanges (DEXs) that use the automated market makers (AMM) algorithm. Readers will gain insights into the inner workings of these innovative trading platforms and develop a DEX smart contract that leverages AMM principles, enabling trustless and permissionless trading of digital assets.

Chapter 17, *Token-Gated Applications*, explores the concept of token-gated applications, which restrict access based on ownership of specific tokens. Readers will learn how to create a signature using an Ethereum wallet, deploy an NFT smart contract, develop a token-gated back-end application, and build a front-end application that enables users to log in securely using their Ethereum accounts and token holdings.

To get the most out of this book

For writing smart contracts, you need Vyper 0.3.10. All code examples will not work with Vyper 0.4. *So, you need to install a specific version of Vyper, which is not the latest version of Vyper*. However, please see adjustments in the GitHub repository if you want to use Vyper 0.4. Other than Vyper, please use the latest version of software / libraries. If there are incompatibilities, please see the GitHub repository. The code has been tested on Ubuntu Linux and MacOS. It's not been tested on Windows, but it should work on Windows. The installation procedures of software on Windows are not covered in the book. But please see the GitHub repository for additional help.

Software/hardware covered in the book

Python (minimum version 3.10 required), Vyper 0.3.10 (versions 0.4.x and above will not work), Ape Framework 0.7.23 (versions 0.8.x and above will not work), modern browsers (Mozilla Firefox, Chrome, etc), Remix, Ganache, Hardhat, Geth, Web3.py, PySide 6, Kubo, aioipfs, Alchemy, Infura, Django, FastAPI, Node.js, Pnpm, React, Wagmi, MetaMask

Operating system requirements: Windows, macOS, or Linux

All software is free to use. For SaaS like Infura or Alchemy, you can use the free tier.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

If you want to deploy smart contracts to production blockchains, such as Ethereum mainnet, then you need to buy or acquire ETH which costs real money!

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition`. If there's an update to the code, it will be updated in the GitHub repository.

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

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: " You want to put the Vyper files inside the `contracts` directory alongside `1_Storage.sol`, `2_Owner.sol`, and `3_Ballot.sol`. Make sure the `contracts` directory is selected by right-clicking it."

A block of code is set as follows:

```
from ape import accounts, project
import os


def main():
    password = os.environ["MY_PASSWORD"]
    dev = accounts.load("dev")
    dev.set_autosign(True, passphrase=password)
    contract = project.SimpleStorage.deploy(sender=dev)
    num_value = contract.retrieve.call()
    print(f"The num value is {num_value}")
```

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

```
(.venv) $ export VOTER_PASSWORD=youraccountpassword
(.venv) $ export VOTER_ACCOUNT=voter1
(.venv) $ export PROPOSAL=0
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Click **contracts** under **FILE EXPLORER** and click **1_Storage.sol**."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Hands-On Blockchain for Python Developers - Second Editon* , we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781805121367`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Blockchain and Smart Contract

In this section, you'll get an overview of the world of decentralized applications. This part begins by introducing the core concepts and principles of blockchain technology, followed by an in-depth exploration of smart contracts – the self-executing programs that enable trustless and automated transactions on the blockchain. Through hands-on examples, you'll gain practical experience in implementing smart contracts using the Vyper language, setting the stage for building secure and robust decentralized applications.

This section has the following chapters:

- *Chapter 1*, *Introduction to Blockchain Programming*
- *Chapter 2*, *Smart Contract Fundamentals*
- *Chapter 3, Using Vyper to Implement a Smart Contract*

1
Introduction to Blockchain Programming

In this book, we'll learn about **blockchain programming** so that you can become a force to be reckoned with when finding blockchain opportunities. To achieve this, you need to begin by understanding blockchain technology and what it entails. In this chapter, we will learn what blockchain technology is. We'll delve into questions such as, how does blockchain empower **Bitcoin** and **Ethereum**? By the end of this chapter, we will get an intuitive understanding of blockchain technology, and we should be able to replicate some of the basic functions behind blockchain.

The following topics will be covered in this chapter:

- The rise of cryptocurrency and blockchain
- Blockchain technology
- Cryptography
- The hashing function
- Consensus
- Coding on the blockchain

Technical requirements

To follow this chapter, you need one of the more recent versions of **Linux** and **Python** – no older than 3.8. You could try implementing what will be covered in this chapter on other platforms, such as Mac or Windows, but Linux is recommended. You could also run Linux with virtualization software such as **VirtualBox**. We will use **Ubuntu Linux 22.04 LTS** and **Python 3.10**.

The rise of cryptocurrency and blockchain

Assuming that you haven't lived a secluded life as a hermit on a mountain, you have probably heard all about **cryptocurrency**, especially Bitcoin. Thus, you needn't have to look far to hear about the soaring popularity of this topic, its terminology, and its growth in value.

Blockchain was regarded as the technology that would bring the dawn of a new era of justice and prosperity for mankind. It would democratize wealth. It would take the power away from the oligarchy and give it back to the people. It would protect the data of the people.

Blockchain started to be used as a payment solution without the middleman, namely Bitcoin. Then, people found out that blockchain has some other interesting properties.

First, it is transparent, meaning people can audit it to check whether there is money laundering going on or not. Second, to some extent, it gives users privacy, which can be used to avoid profiling.

Then, after **Ethereum** was released, people suddenly became creative with how to apply blockchain in real life – from creating a token to represent the ownership of something, such as an autonomous organization or payment with full privacy, to digital assets that cannot be duplicated (unlike MP3 files).

Blockchain technology

Most people know Bitcoin exists because of blockchain.

But what is blockchain?

It is an *append-only database* that consists of blocks that are linked by **hashing**. Here, each block contains many transactions of transferring value (but could be other things) between participants secured by cryptography; a consensus between many nodes that hold an identical database decides on which new block is to be appended next.

You don't have to understand the definition at this point; those are a lot of words to chew on! First, I'll explain blockchain so that you can adjust to this new knowledge as we move through this book.

Going back to the definition of blockchain, we can summarize the definition as an append-only database. *Once you put something into the database, it cannot be changed; there is no undo button*. We'll talk about the ramifications of this feature in *Chapter 2*, *Smart Contract Fundamentals*. This definition entails many things and opens up a whole new world.

So, what can you put into this append-only database?

It depends on the cryptocurrency. For Bitcoin, you can store the transactions of transferring value. For example, Nelson sends one Bitcoin to Dian. However, we accumulate many transactions into one block before appending them to the database. For Ethereum, the things that you can put into the append-only database are richer. This not only includes transferring value – it could also be a change of state. I am referring to state here in a general sense. For example, a queue for buying a ticket for a

show can have a state. This state can be empty or full. Similar to Bitcoin, in Ethereum, you can gather all the transactions before appending them together in this append-only database.

To make this clearer, we put all these transactions into the block before appending them to the append-only database. Aside from the list of transactions, we store other things in this block, such as the time when we append the block into the append-only database, the target's difficulty (don't worry if you don't know about this), and the parent's hash (I'll explain this shortly), among many other things.

Now that you understand the **block element** of the blockchain, let's look at the **chain element**. As explained previously, aside from the list of transactions, we also put the parent's hash in the block. But for now, let's just use a simple ID to indicate the parent instead of using a hash. The **parent ID** is just the previous block's ID. Here, think of the stack. In the beginning, there is no block. Instead, we put **Block A**, which has three transactions: *Transaction 1*, *Transaction 2*, and *Transaction 3*. Since **Block A** is the first block, it has no parent. Then, we apply **Block B** to **Block A**, which consists of two transactions: *Transaction 4* and *Transaction 5*. **Block B** is not the first one in this blockchain. Consequently, we set the parent section in **Block B** as the **Block A** ID because **Block A** is the parent of **Block B**. Then, we put **Block C** in the blockchain, which has two transactions: *Transaction 6* and *Transaction 7*.

The parent section in **Block C** would be the **Block B** ID, and so on. To simplify things, we increment the ID from 0 by 1 for every new block. *Figure 1.1* shows three blocks connected by the parent ID field:

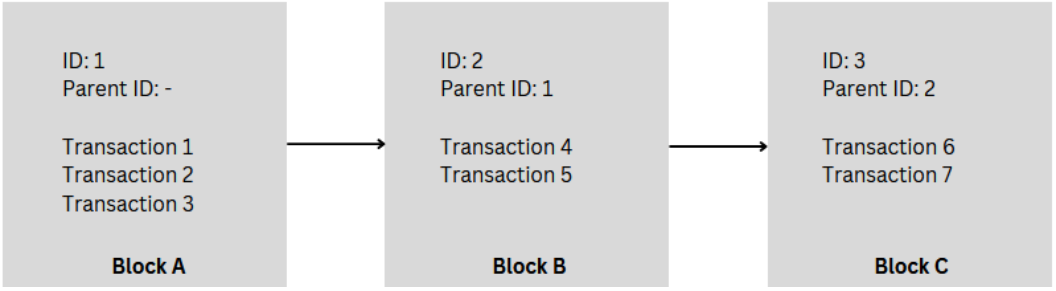


Figure 1.1: A linked list of Block A, Block B, and Block C

There are seven transactions linked together in three blocks. *Transaction 1*, *Transaction 2*, and *Transaction 3* happened at the same time because they are located in **Block A**. *Transaction 4* and *Transaction 5* happened after transactions in **Block A**, and so on.

As an example, let's implement a database to record the history of what people like and hate. This means that when you said you liked cats at one point in history, you won't be able to change that history. You may add new history when you change your mind (for example, if you then hate cats), but that won't change the fact that you liked them in the past. So, we can see that you liked cats in the past, but now, you hate them. We want to ensure this database is full of integrity and secure against cheating. Look at the following code block:

```
class Block:
    id = None
```

```
    history = None
    parent_id = None

block_A = Block()
block_A.id = 1
block_A.history = 'Nelson likes cat'

block_B = Block()
block_B.id = 2
block_B.history = 'Marie likes dog'
block_B.parent_id = block_A.id

block_C = Block()
block_C.id = 3
block_C.history = 'Sky hates dog'
block_C.parent_id = block_B.id
```

If you studied computer science, you would recognize this data structure, which is called a **linked list**. A linked list includes collection nodes linked together by pointers between nodes. So, each node (except the root node and the last node) has connections to the next node and the previous node.

Now, there is a problem. Say Marie hates Nelson and wants to paint Nelson in a negative light. Marie can do this by changing the history of `block_A`:

```
block_A.history = 'Nelson hates cat'
```

This is unfair to Nelson, who is a huge fan of cats. So, we need to add a way in which only Nelson can write the history of his preferences. We can do this using a **private key** and a **public key**.

Signing data in blockchain

In blockchain, we use two keys to sign data so that we can authenticate a message and protect it from being altered by unauthorized users: a private key and a public key.

The secrecy of the private key is guarded, and it isn't made known to the public. On the other hand, you let the public key be given out to the public. You tell everyone, "*Hey, this is my public key.*"

Let's generate the private key. To do this, we need the *OpenSSL software*. You can install this by running the following command:

```
$ sudo apt-get install openssl
```

The preceding command installs `openssl` on Ubuntu Linux.

So, Nelson generates the private key, which is the `nelsonkey.pem` file. He must keep this key secret. It can be generated like so:

```
$ openssl genrsa -out nelsonkey.pem 1024
```

From the private key, Nelson generates the public key:

```
$ openssl rsa -in nelsonkey.pem -pubout > nelsonkey.pub
```

Nelson can share this public key, `nelsonkey.pub`, with everyone. Now, in the real world, we could set up a simple dictionary of the public key and its owner, as follows:

```
{
'Nelson': 'nelsonkey.pub',
'Marie': 'mariekey.pub',
'Sky': 'skykey.pub'
}
```

Let's look at how Nelson can prove that he is the only one who can make changes to his history.

First, let's create a Python virtual environment:

```
$ python3.10 -m venv .venv
$ source .venv/bin/activate
(.venv) $
```

Next, install the library:

```
(.venv) $ pip install cryptography
```

The last command installed the `cryptography` Python library that we're going to use later.

Here's the Python script that can be used to sign the message. Name this script `verify_message.py` (refer to the code file at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_1/verify_message.py` for the full code):

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization


…
public_key.verify(
```

```
    signature,
    MESSAGE,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
```

When executing this script, nothing will happen, as expected. This means that the message is verified with the `signature` value from the public key. This `signature` can only be created by Nelson because you need the private key to create one. However, to verify the message with `signature`, you only need the public key.

Let's look at a case in which Marie tries to falsify the facts with a script named `falsify_message.py`. Marie tries to put `Nelson hates cat` in the history database, as follows:

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

message = b'Nelson hates cat'
signature = b'Fake Signature'

with open("nelsonkey.pub", "rb") as key_file:
    public_key = serialization.load_pem_public_key(
        key_file.read(),
        backend=default_backend())

public_key.verify(
    signature,
    message,
    padding.PSS(mgf=padding.MGF1(hashes.SHA256()), \
                salt_length=padding.PSS.MAX_LENGTH),
    hashes.SHA256())
```

Let's learn how we can verify that the method works. Nelson calculates the `hash` value from the message and then encrypts it with his private key. The result is a `signature` value. For example, if Sky wants to `verify` the signature, he has the message and the signature. He calculates the hash of the message. Then, he decrypts the `signature` value using the public key. The result is compared to the hash of the message. If it is the same, then everything is well. If not, either the message has been altered or the private key that was used to sign the message is different.

When doing this, you will get the following output:

```
Traceback (most recent call last):
  File "/home/arjuna/Documents/book/chapter_1/falsify_message.py",
line 15, in <module>
    public_key.verify(
  File "/home/arjuna/Documents/book/chapter_1/.venv/lib/python3.10/
site-packages/cryptography/hazmat/backends/openssl/rsa.py", line 571,
in verify
    _rsa_sig_verify(
  File "/home/arjuna/Documents/book/chapter_1/.venv/lib/python3.10/
site-packages/cryptography/hazmat/backends/openssl/rsa.py", line 322,
in _rsa_sig_verify
    raise InvalidSignature
cryptography.exceptions.InvalidSignature
```

So, what does the `signature` value look like? Go back to `verify_message.py` and append this line to the end of the file. Then, run the script again:

```
print(signature)
```

`signature` looks like this:

```
b'A|4=\xae\x99\xdf\xf6H\x9f=\xfb\xa9\x06\xc3K\x81\x9b\xcc_\x8f\xf0@\
x98\xbb$\xba\xa4\x05\xff\x92\x12\x1e\xdb\x81\xe2\xf6\xc1!\x83r\xcd\
xf5\x88?cF\xfa\x14\x91\xd1\'T\xd5\x95\xeb}\'B")\xcds\x9a\x8d\xe3\x1bL\
xcel\xf0\xd8(\x92\x8e\xbc/\x82\x05\x7f\xd4$G\x06\x107l\x80\xf0\xf8v\
xf9\r\x8a\x15\t\r\xe09(\x84L\xe1!\xbd\xf0\xc1\x183\x0e\xfd\r:x\x14)M\
x14\x18M&\xf6\x06\n\xf1\x96\t\xfa'
```

Yours will be different because we have different private keys, but it should look similar.

Every message has a different `signature`, and Marie can't guess the `signature` value to falsify the message.

> **Note**
>
> So, with the private and public keys, we can verify whether the message is from someone who's been authorized, even if we communicate on an unsecure channel.

eAs shown in the following figure, with the private key, Nelson can create a signature that's unique to the message it tries to sign:

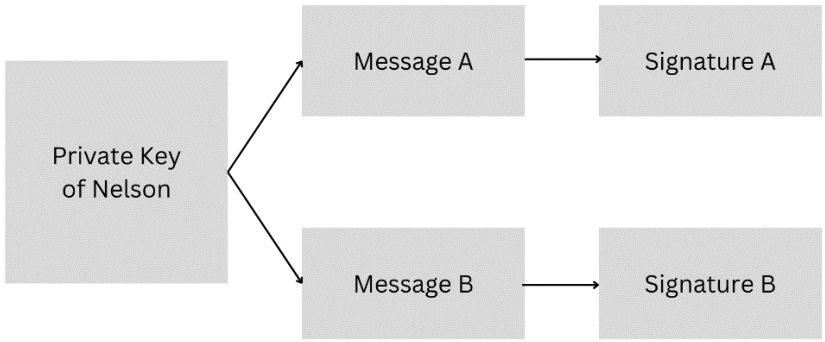


Figure 1.2: The signature comes from a private key signing message

Everyone in the world who has Nelson's public key can verify that Nelson did indeed write **Message A**. Nelson can prove he wrote **Message A** by showing **Signature A**. Everyone can take those two inputs and verify the truth, as shown in *Figure 1.3*:

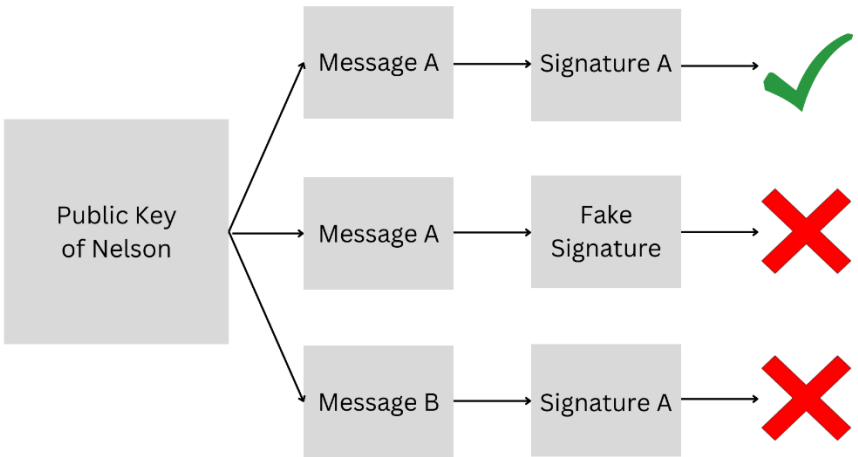


Figure 1.3: Checking the validity of signatures

Everyone has the public key of Nelson and the signature from Nelson, **Signature A**. Nelson wrote two messages: **Message A** and **Message B**.

So, to validate whether or not it is Nelson who wrote `Nelson likes cat`, input the following code (refer to `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_1/validate_message.py` for the full code):

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

def fetch_public_key(user):
    with open(user.decode('ascii') + "key.pub", "rb") as key_file:
        public_key = serialization.load_pem_public_key( \
           key_file.read(),\
           backend=default_backend())
    return public_key

# Message coming from user
message = b"Nelson likes cat"

# Signature coming from user
signature = b'A|4=\xae\x99\xdf\xf6H\x9f=\xfb\xa9\x06\xc3K\x81\x9b\
xcc_\x8f\xf0@\x98\xbb$\xba\xa4\x05\xff\x92\x12\x1e\xdb\x81\xe2\xf6\
xc1!\x83r\xcd\xf5\x88?cF\xfa\x14\x91\xd1\'T\xd5\x95\xeb}\'B")\xcds\
x9a\x8d\xe3\x1bL\xcel\xf0\xd8(\x92\x8e\xbc/\x82\x05\x7f\xd4$G\x06\
x107l\x80\xf0\xf8v\xf9\r\x8a\x15\t\r\xe09(\x84L\xe1!\xbd\xf0\xc1\x183\
x0e\xfd\r:x\x14)M\x14\x18M&\xf6\x06\n\xf1\x96\t\xfa'

user = message.split()[0].lower()
# fetch public key from Nelson
public_key = fetch_public_key(user)
# … verify the message like before
public_key.verify(
    signature,
    message,
    padding.PSS(mgf=padding.MGF1(hashes.SHA256()),\
                salt_length=padding.PSS.MAX_LENGTH),\
    hashes.SHA256())
```

Now that you've understood the signature, it's time to build something bigger, which is blockchain!

From linked list to blockchain

We can be at peace as we now know that only Nelson can write `Nelson likes cats` or `Nelson hates cats`. However, to make the tutorial code short, we won't integrate the validation using the private key and the public `key`. We assume only authorized people can write the history in the block. Look at the following code block:

```
>>> block_A.history = 'Nelson likes cat'
```

When that happens, we assume it is Nelson who wrote that history. So, what's the problem in recording data with a linked list?

The problem is that the data can be altered easily. Say Nelson wants to be a senator. If many people in his district don't like cats, they may not be happy with the fact that Nelson likes them. Consequently, Nelson wants to alter history:

```
>>> block_A.history = 'Nelson hates cat'
```

Just like that, history has changed. We can avoid this way of cheating by recording all history on the block every day. So, when Nelson alters the database, we can compare the data in the blockchain today to the data in the blockchain yesterday. If it is different, we can confirm that something fishy is happening. That method could work, but let's see if we can produce something better.

Let's upgrade our linked list to blockchain. To do this, we'll add a new property in the `Block` class, which is the parent's `hash` value:

```
import hashlib
import json


class Block:
    id = None
    history = None
    parent_id = None
    parent_hash = None

block_A = Block()
block_A.id = 1
block_A.history = 'Nelson likes cat'

block_B = Block()
block_B.id = 2
block_B.history = 'Marie likes dog'
block_B.parent_id = block_A.id
block_B.parent_hash = hashlib.sha256(json.dumps(block_A.__dict__).
encode('utf-8')).hexdigest()
```

```
block_C = Block()
block_C.id = 3
block_C.history = 'Marie likes dog'
block_C.parent_id = block_B.id
block_C.parent_hash = hashlib.sha256(json.dumps(block_B.__dict__).
encode('utf-8')).hexdigest()
```

Let's demonstrate what the `hashlib` function does:

```
>>> from simple_blockchain import block_B
>>> import json, hashlib
>>> print(block_B.__dict__)
{'id': 2, 'history': 'Marie likes dog', 'parent_id': 1, 'parent_hash':
'69a1db9d3430aea08030058a6bd63788569f1fde05adceb1be6743538b03dadb'}
>>> print(json.dumps(block_B.__dict__))
{"id": 2, "history": "Marie likes dog", "parent_id": 1, "parent_hash":
"69a1db9d3430aea08030058a6bd63788569f1fde05adceb1be6743538b03dadb"}
>>> print(json.dumps(block_B.__dict__).encode('utf-8'))
b'{"id": 2, "history": "Marie likes
dog", "parent_id": 1, "parent_hash":
"69a1db9d3430aea08030058a6bd63788569f1fde05adceb1be6743538b03dadb"}'
>>> print(hashlib.sha256(json.dumps(block_B.__dict__).
encode('utf-8')))
<sha256 _hashlib.HASH object @ 0x7f53c11e95f0>
>>> print(hashlib.sha256(json.dumps(block_B.__dict__).
encode('utf-8')).hexdigest())
ea6043f03881a1808cd073f65b73cdf246d6036d12bf236d18c532717ec05d06
```

Let's change the history of `block_A` with the following code:

```
>>> from simple_blockchain import block_A
>>> block_A.history = 'Nelson hates cat'
```

Again, history has changed just like that. However, this time, there's a twist. We can verify that this change has occurred by printing the original parent's `hash` of `block_C`:

```
>>> from simple_blockchain import block_C
>>> print(block_C.parent_hash)
ea6043f03881a1808cd073f65b73cdf246d6036d12bf236d18c532717ec05d06
Now, let's recalculate the parent's hash of each block:
>>> block_B.parent_hash = hashlib.sha256(json.dumps(block_A.__dict__).
encode('utf-8')).hexdigest()
>>> block_C.parent_hash = hashlib.sha256(json.dumps(block_B.__dict__).
encode('utf-8')).hexdigest()
>>> print(block_C.parent_hash)
95e67747e67d6e37d2533734a6984d69695a6bec655c6d161224cd000b0f5fa8
```

These blocks are different. By looking at these, we can be very sure that history has been altered. Consequently, Nelson would be caught red-handed. Now, if Nelson wants to alter history without getting caught, it isn't enough to change the history in `block_A` anymore. Nelson needs to change all the `parent_hash` properties in every block (except `block_A` of course). This is a tall order.

> **Note**
> With only three blocks, Nelson needs to change two `parent_hash` properties. With 1,000 blocks, Nelson needs to change 999 `parent_hash` properties!

But hashing alone is not enough to create a cryptocurrency. There's the word *crypto* in cryptocurrency. That word comes from *cryptography*.

Cryptography

The most popular use of blockchain is to create cryptocurrency. As the word crypto is in cryptocurrency, you would expect that you need to master cryptography to become a blockchain programmer. That isn't true. You only need to know two things about cryptography:

- Private keys and public keys (asymmetric cryptography)
- Hashing

We explained these previously. You don't need to know how to design a hashing algorithm or private key and public key algorithm. You only need to get an intuitive understanding of how they work and the implications of these technologies.

Private keys and public keys enable decentralized accounts. In a normal application, you have a username and password. These two fields enable someone to access their account. However, having a private key and a public key enables someone to have an account in a decentralized manner.

For hashing, it is a one-way function, meaning that given an input, you can get the output easily. However, given an output, you couldn't get the input. A simple version of a one-way function is shown here:

$f(x,y)\ x + y$

This is an addition process. If I tell you one of the outputs of this function is `999`, and I ask you what the inputs are, you won't be able to guess the answer. It could be anything from `1` and `998` to `500` and `499`. A hashing function is something like that. The algorithm is as clear as the sky (you can read the algorithm of any hashing function on the internet). But it is hard to reverse the algorithm.

So, all you need to know about hashing is that given any input, you'll get the following `SHA-256` output (in hexadecimal): `c96c6d5be8d08a12e7b5cdc1b207fa6b2430974c86803d8891675e76fd992c20`.

If you don't know the input, you can't get the input based on this output alone. Say you know the input; it is very prohibitive to find another input that produces the same output. We wouldn't even know whether such an input exists or not.

This is all you need to know about cryptography when you become a blockchain developer. But that is only true if you become a certain type of blockchain developer, who creates a program on top of Ethereum.

The internet and blockchain are built on the foundation of asymmetric cryptography. But it's good to know another kind of cryptography as well, which is **symmetric cryptography**.

Symmetric and asymmetric cryptography

Symmetric cryptography uses the same key between the sender and the receiver. This key is used to encrypt and decrypt a message. For example, say you want to create an encryption function to encrypt text. Symmetric cryptography could be as simple as adding `5` to the text to be encrypted. If `A` (or `65` in ASCII) is the text to be encrypted, then this encryption function will add `5` to `65`. The encrypted text would be `F` (or `71` in ASCII). To decrypt it, you just subtract `5` from the encrypted text, `F`.

Asymmetric cryptography is a different beast. There are two keys: a public key and a private key. They are linked with a special mathematical relationship. If you encrypt a message with a public key, you can only decrypt it with a private key. If you encrypt a message with a private key, you can only decrypt it with a public key. There is no straight relationship as with symmetric keys (adding and subtracting the same number) between a public key and a private key. There are a couple of asymmetric cryptography algorithms. I will explain the easiest one, the **RSA algorithm**.

Generate two prime numbers, called `p` and `q`. They should be big numbers (with hundreds of digits), but for this example, we'll choose small numbers: `11` and `17`. These are your private key values. Don't let someone know these numbers:

```
n = p x q
```

`n` is a composite number. In our case, `n` is `187`.

Then, we find the number for `e`, which should be relatively prime, with `(p-1) x (q-1)`:

```
(p-1) x (q-1) = 160
```

Relatively prime means `e` and `(p-1) x (q-1)` cannot be factorized with any number except `1`. There is no number other than `1` that we can divide without a remainder. So, `e` is `7`. However, `e` can be `11` as well. For this example, we'll choose `7` for `e`.

`e` and `n` are your public key values. You can tell these numbers to strangers you meet on the bus, your grandma, your friendly neighbor, or your date.

Let's say the message we want to encrypt is `A`. In the real world, encrypting a short message like this isn't safe. We must pad the short message. So, `A` would be something like `xxxxxxxxxxxxxxxxxxxA`. If you check out the previous script for encrypting a message that was provided earlier in this chapter, you will see that there is a padding function. But for this example, we won't pad the message.

The encryption function is shown here:

```
encrypted_message = message exp e (mod n)
```

So, `encrypted_message` would be `65 ** 7 % 187 = 142`.

Before we can decrypt the message, we need to find the `d` number:

```
e x d = 1 (mod (p-1) x (q-1))
```

Here, `d` is `23`.

The decryption function is shown here:

```
decrypted_message = encrypted_message exp d mod n
```

So, `decrypted_message` would be `142 ** 23 % 187 = 65`. `65` in ASCII is `A`.

`x exp y mod n` is easy to calculate, but finding the `y` root of integer module `n` is hard. We call this trapdoor permutation. Factorization of `n` to find `p` and `q` is hard (generating a private key from a public key). However, finding `n` from `p` and `q` is easy (generating a public key from a private key). These properties enable asymmetric cryptography.

> **Note**
>
> Compared to symmetric cryptography, asymmetric cryptography enables people to communicate securely without needing to exchange keys first. You have two keys (a private key and a public key). You throw the public key out to anyone. All you need to do is protect the secrecy of the private key. The private key is like a password to your Bitcoin/Ethereum account. Creating an account in any cryptocurrency is just generating a private key. Your address (or your username in cryptocurrency) is derived from the public key. The public key itself can be derived from the private key.

An example of Bitcoin's private key in **Wallet Import Format** (**WIF**) is `5K1vbDP1nxvVYPqdKB5wCVpM3y99MzNqMJXWTiffp7sRWyC7SrG`. It has 51 hexadecimal characters. Each character can have 16 combinations. So, the amount of private keys is `16 ^ 51 = 25711008708143844408671393477458601640355247900524685364822016` (it's not exactly this amount, because the first number of a private key in Bitcoin is always `5` in mainnet, but you get the idea). That is a vast number. So, the probability of someone finding another account that is filled with Bitcoin already when generating a private key with a strong random process is

extremely low. However, the kind of account that's generated by a private key and public key doesn't have a reset password feature.

> **Tip**
> If someone sends Bitcoin to your address and you forget your private key, then it has gone for good. So, while your public key is recorded on the blockchain that's kept in every Bitcoin node, people aren't going to get the private key.

The hashing function

Hashing is a function that takes an input of any length and turns it into a fixed-length output. To make this clearer, we can look at the following code example:

```
>>> import hashlib
>>> hashlib.sha256(b"hello").hexdigest()
'2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824'
>>> hashlib.sha256(b"a").hexdigest()
'ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb'
>>> hashlib.sha256(b"hellohellohellohello").hexdigest()
'25b0b104a66b6a2ad14f899d190b043e45442d29a3c4ce71da2547e37adc68a9'
```

As you can see, the length of the input can be `1`, `5`, or even `20` characters, but the output will always be the length of 64 hexadecimal numeric characters. The output looks scrambled, and there is no apparent link between the input and the output. However, if you give the same input, it will give the same output every time:

```
>>> hashlib.sha256(b"a").hexdigest()
'ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb'
>>> hashlib.sha256(b"a").hexdigest()
'ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb'
```

If you changed the input by even just a character, the output would be different:

```
>>> hashlib.sha256(b"hello1").hexdigest()
'91e9240f415223982edc345532630710e94a7f52cd5f48f5ee1afc555078f0ab'
>>> hashlib.sha256(b"hello2").hexdigest()
'87298cc2f31fba73181ea2a9e6ef10dce21ed95e98bdac9c4e1504ea16f486e4'
```

Now that the output has a fixed length, which is 64 in this case, there will be two different inputs that have the same output.

> **Note**
>
> Here's the interesting thing: it is very prohibitive to find two different inputs that have the same output as this hashing function. Even if you hijack all the computers in the world and make them run the hashing computation, it is unlikely that you would ever find two different inputs with the same output.

Not all hashing functions are safe, though. `SHA-1` died in 2017. This means that people can find two different long strings that have the same output. In this example, we will use `SHA-256`.

The output of the hashing function can be used as a digital signature. Imagine that you have a string with a length of 10 million (say you are writing a novel), and to make sure this novel is not tampered with, you tell all your potential readers that they have to count the 10 million characters to ensure that the novel isn't corrupted. Nobody would do that. But with hashing, you can publish the output validation with only 64 characters (through Twitter, for example) and your potential readers can hash the novel that they buy/download and compare them to make sure that their novel is legit.

So, we add the parent's `hash` in the `block` class. This way, we keep the digital signature of the parent's `block` in our `block`. This means that if we, out of sheer mischief, change the content of any block, the parent's `hash` in any child's block will be invalid, and you will get caught red-handed.

But can't you change the parent's `hash` of the children's `block` if you want to alter the content of any `block`? Yes, you can. However, the process of altering the content becomes more difficult. In such a case, you have to take two steps. Now, imagine you have 10 `blocks`, and you want to change the content in the first `block`.

In this case, you must change the parent's `hash` in its immediate child's `block`. But, alas, there are unseen ramifications with this. Technically speaking, the parent's `hash` in its immediate child is a part of the content in that block. That would mean that the parent's `hash` in its child (the grandchild of the first block) would be invalid.

Now, you must change that grandchild's parent's `hash`, but this affects the subsequent `block`. Now, you must change all blocks' parent's hashes. For this, 10 steps need to be taken. Using a parent's `hash` makes tampering much more difficult.

Now that you understand hashing, it's time to learn the innovation built on top of it. This brings us to **proof of work**.

Proof of work

Words are cheap, actions are worthy. Hence, we should keep in mind that while everyone can do hashing, not everyone is willing to do hashing a lot of times. To make sure someone is worthy of securing data in the blockchain, they must be willing to do a lot of hashing processes. This is the foundation of blockchain technology.

So, we have three participants in this case: `Nelson`, `Marie`, and `Sky`. But there is another type of participant too: the one who writes into the blockchain. This participant is called – in blockchain parlance – the **miner**. To put the transaction into the blockchain, the miner is required to do some work first.

Previously, we had three blocks (`block_A`, `block_B`, and `block_C`), but now, we have a candidate block (`block_D`) that we want to add to the blockchain, as follows:

```
block_D = Block()
block_D.id = 4
block_D.history = 'Sky loves turtle'
block_D.parent_id = block_C.id
```

But instead of adding `block_D` to the blockchain just like that, we require the miner to do some puzzle work. We serialize that block and ask the miner to apply an extra string, which, when appended to the serialization string of that block, will show the hash output with at least five zeros in the front if it is hashed.

Those are a lot of words to chew on. First things first, we must serialize the block:

```
import json
block_serialized = json.dumps(block_D.__dict__).encode('utf-8')
print(block_serialized)
b'{"history": "Sky loves turtle", "parent_id": 3, "id": 4}'
```

If the serialized block is hashed, what does it mean if we want the `hash` output to have at least five zeros at the front? It means that we want the output to look like this:

```
00000aa21def23ee175073c6b3c89b96cfe618b6083dae98d2a92c919c1329be
```

Alternatively, we want it to look like this:

```
00000be7b5347509c9df55ca35d27091b41a93acb2afd1447d1cc3e4b70c96ab
```

So, the puzzle is something like this:

```
string serialization + answer = hash output with (at least) 5 leading
zeros
```

The miner needs to guess the correct answer. If this puzzle is converted into Python code, it would look something like this:

```
answer = ?
input = b'{"history": "Sky loves turtle", "parent_id": 3, "id": 4}' +
answer
output = hashlib.sha256(input).hexdigest()
```

```
// output needs to be
00000????????????????????????????????????????????????????????????
```

The output of the `hexdigest` method is slightly random. But we need the output to have `00000` at the start. The key is to find what the value of `answer` will give this output.

So, how could the miner solve a problem like this? We can use brute force:

```
import hashlib
payload = b'{"history": "Sky loves turtle", "parent_id": 3, "id": 4}'
for i in range(10000000):
          nonce = str(i).encode('utf-8')
          result = hashlib.sha256(payload + nonce).hexdigest()
          if result[0:5] == '00000':
              print(i)
              print(result)
              break
```

The result would be as follows:

```
184798
00000ae01f4cd7806e2a1fccd72fb18679cb07ede3a2a7ef028a0ecfd4aec153
```

This means that the answer is `184798`, or the hash output of `{"history": "Sky loves turtle", "parent_id": 3, "id": 4}184798` is the one that has five leading zeros. In that simple script, we iterate from `0` to `9999999` and append that into the input. This is a naive method, but it works. Of course, you could also append characters other than numbers, such as `a`, `b`, or `c`.

Now, try to increase the number of leading zeros to six, or even 10. In this case, can you find the hash output? If there is no output, you could increase the range limit from `10000000` to an even higher number, such as `1000000000000`. Once you get an appreciation of the demanding work that goes into this, try to comprehend this: at the time of writing, Bitcoin requires around 18 leading zeros in the hash output. The number of leading zeros is not static and changes according to the situation (but you don't need to worry about this).

So, why do we need proof of work? Before we answer this, let's look at the idea of consensus.

Consensus

As we can see, the hashing function makes history tampering hard, but not too hard. Even if we have a blockchain that consists of `1000` blocks, it would be trivial to alter the content of the first block and change the `999` parent hashes on the other blocks with recent computers. So, to ensure that bad people cannot alter history (or at least make it hard), we must distribute this append-only database to everyone who wants to keep it (let's call them miners). Say there are 10 miners. In this case, you cannot just alter the blockchain in your copy because the other nine miners would scold you, saying something like *Hey, our records say history A but your record says B. In this case, the majority wins*.

However, consensus is not just a case of choosing which blockchain has been chosen by the majority. The problem starts when we want to add a new block to the blockchain.

Where do we start? How do we do it?

The answer is that we broadcast. When we broadcast the candidate block that contains a new transaction, it won't reach every miner at the same time. You may reach the miner who stands beside you, but it will require time for your message to reach the miner who's standing far away from you.

Here's where it gets interesting: the miner that's standing far away from you may receive another new candidate block first. So, how do we synchronize all these things and make sure that the majority will have the same blockchain? The simple rule is to choose the longest chain. So, if you are a miner in the middle, you may receive two different candidate blocks at the same time, as shown in the following figure:

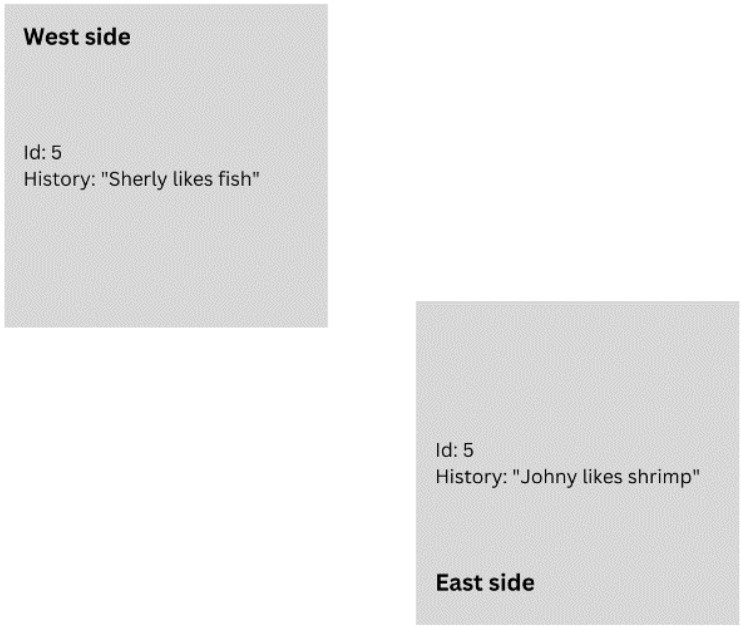


Figure 1.4: Competing messages from the west side and the east side

Figure 1.4 tells a story of two sides proposing a different history for the same ID.

You get the following from the west side:

```
block_E = Block()
block_E.id = 5
block_E.history = 'Sherly likes fish'
block_E.parent_id = block_D.id
```

On the other hand, you get the following from the east side:

```
block_E = Block()
block_E.id = 5
block_E.history = 'Johny likes shrimp'
block_E.parent_id = block_D.id

block_F = Block()
block_F.id = 6
block_F.history = 'Marie hates shark'
block_F.parent_id = block_E.id

block_G = Block()
block_G.id = 7
block_G.history = 'Sarah loves dog'
block_G.parent_id = block_F.id
```

At this point, we can get rid of the west-side version of the blockchain because we chose the longer version.

Here comes the problem. Say Sherly hates sharks but she wants to get votes from a district where most people only vote for a candidate who loves sharks. To get more votes, Sherly broadcasts a block containing the following lie:

```
block_E = Block()
block_E.id = 5
block_E.history = 'Sherly loves shark'
block_E.parent_id = block_D.id
```

All is fine and dandy. The voting session takes 1 day. After this time passes, the blockchain gets another two blocks:

```
block_E = Block()
block_E.id = 5
block_E.history = 'Sherly loves shark'
block_E.parent_id = block_D.id

block_F = Block()
block_F.id = 6
block_F.history = 'Lin Dan hates crab'
block_F.parent_id = block_E.id
```

The following figure illustrates the three blocks:

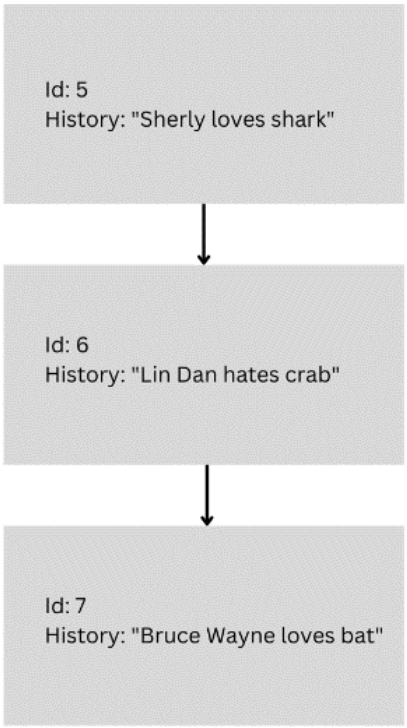


Figure 1.5: Three linked boxes

Now, Sherly needs to get votes from another district where most people only vote for candidates who hate sharks.

So, how can Sherly tamper with the blockchain to make this work in her favor?

Sherly could broadcast four blocks!

```
block_E = Block()
block_E.id = 5
block_E.history = 'Sherly hates shark'
block_E.parent_id = block_D.id

block_F = Block()
block_F.id = 6
block_F.history = 'Sherly loves dog'
block_F.parent_id = block_E.id

block_G = Block()
```

```
block_G.id = 7
block_G.history = 'Sherly loves turtle'
block_G.parent_id = block_F.id

block_H = Block()
block_H.id = 8
block_H.history = 'Sherly loves unicorn'
block_H.parent_id = block_G.id
```

The following figure illustrates these four blocks:

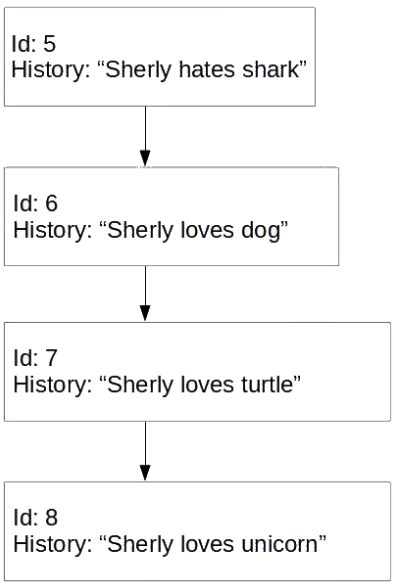


Figure 1.6: Four blocks linked together

The miner will choose the blockchain from Sherly instead of the previous blockchain they kept, which contains the history of `Sherly loves sharks`. So, Sherly has been able to change the history. This is what we call a double-spending attack.

We can prevent this through proof of work (an incentive for adding blocks). We explained proof of work earlier in this chapter, but we haven't explained the incentive system yet. An incentive means that if the miner successfully adds a new block to the blockchain, the system gives them a digital reward. We can integrate it into the code as follows:

```
import hashlib
payload = b'{"history": "Sky loves turtle", "parent_id": 3, "id": 4}'
for i in range(10000000):
nonce = str(i).encode('utf-8')
```

```
result = hashlib.sha256(payload + nonce).hexdigest()
if result[0:5] == '00000':
// We made it, time to claim the prize
reward[miner_id] += 1
print(i)
print(result)
break
```

This code is simple enough that it can be run with only a simple computer. But in the real world, to be a miner, you need to have **application-specific integrated circuit** (**ASIC**) machines. The hardware is created specifically for this purpose.

Blockchain as a technology that prevents cheating

If Sherly wants to alter the history (by replacing some blocks), she needs to spend some resources by solving four puzzles in a short time. By the time she finishes doing this, the blockchain that's kept by most miners would have added more blocks, making it longer than Sherly's blockchain.

This is the case because most miners want to get that reward we spoke of in the most efficient manner possible. To do this, they must get a new candidate block, work hard to find the answer in proof of work, and then add it to the longest chain as quickly as possible. But why do they want to add it to the longest chain and not another chain? This is because it secures their reward.

Say we have two versions of the blockchain. One has three blocks, while the other has eight blocks. The most sensible way to add a new block is to add it to the blockchain that has eight blocks. If someone adds it to the blockchain that has three blocks, it is more likely to get discarded. Consequently, the reward would be taken away from the miner. The longest chain attracts the most miners anyway, and you want to be in the blockchain version that is kept by more people.

Some miners could persist in adding the block to the blockchain with three blocks, while other miners could also persist in adding the block to the blockchain with eight blocks. We call this a hard fork. Most of the time, miners will stick to the blockchain that has the longest chain.

To change history, Sherly will need to outgun at least more than 50% of the miners, which is impossible. The older the block, the more secure the history in that block is. Say one person needs 5 minutes to do the puzzle work. In this case, to replace the last five blocks in the blockchain, Sherly needs more than 25 minutes (because Sherly needs at least six blocks to convince miners to replace the last five blocks in their blockchain). But in those 25 minutes, other miners would keep adding new blocks to the most popular blockchain. So, when 25 minutes have passed, the most popular blockchain would have gained an additional five blocks! Maybe the miners take a nap for an hour and don't add any more blocks. In this case, Sherly could accumulate six blocks to tamper with the most popular blockchain. However, the incentive that's embedded in the blockchain keeps the miners awake 24/7 as they want to get the reward as much as possible. Consequently, it is a losing battle for Sherly.

Bitcoin uses proof of work. In the past, Ethereum used proof of work as well. But today, Ethereum uses proof of stake. So, let's see how it differs from proof of work.

Proof of stake

As mentioned previously, Ethereum does not use proof of work anymore. From September 2023 onward, Ethereum uses proof of stake. It doesn't involve brute-force guessing input for the hashing function anymore. It uses Ethereum currency as a stake. At the time of writing, to become a validator (akin to miners in proof of work), you need to stake 32 ETH.

> **Note**
>
> If, as a validator, you cheat and other validators confirm it, your stake could be gone. The stake forces you to be honest.

The reasons Ethereum switched to proof of stake are as follows:

- It is more environmentally friendly
- It is more affordable for people who want to be validators
- It is scalable

To imagine how proof of stake works, consider that there are 10 validators, and each of them stakes 32 ETH.

The Ethereum network will randomly choose a validator to propose a new block. Let's say that for this occasion, the network chooses Validator 2, which will find transactions on a pool. These transactions are from people who want to create transactions, such as you sending ETH to your grandma or your neighbor creating smart contracts. Validator 2 will execute these transactions and make sure that they are valid (if you send 2 ETH to your grandma, you must have at least 2 ETH). Then, it will create a block containing these transactions and propose it. Other validators will also check the block to ensure Validator 2 doesn't cheat. If other validators confirm this, the block will be appended to the blockchain.

Then, the Ethereum network chooses another validator randomly. This time, it chooses Validator 7. However, Validator 7 is offline. Because Validator 7 fails to uphold its responsibility, it gets a penalty.

At this point, the Ethereum network chooses another validator, Validator 5. The cycle repeats.

This is the basic idea of proof of stake. Of course, the real-world implementations are more complicated. If you want to register as a validator, there is a queue. This is to prevent dishonest validators from swarming Ethereum networks. When the Ethereum network chooses a validator to propose a block, it isn't 100% random because it considers the age of the validator in the Ethereum network and the last time this validator became the proposer validator.

The original intention of blockchain technology is to keep data safe from tampering or cheating. This is the spirit of Bitcoin software. However, over time, the technology has become more expansive. Blockchain has become the foundation of computation that's free from cheating. This is the spirit of Ethereum. Ethereum is a platform for computation or programming that's transparent and secure from tampering.

Coding on the blockchain

At the time of writing, the two most popular cryptocurrencies are Bitcoin and Ethereum.

> **Note**
>
> If you ask someone who knows a lot about cryptocurrencies a simple question, you may get the following answer: Bitcoin is just for sending money, but you can create a program on Ethereum. The program can be tokens, auctions, or escrow, among many other things. But that is a half-truth.

You can also create a program via Bitcoin. Usually, people call this program a script. It is necessary to provide a script in a Bitcoin transaction. A transaction in Bitcoin can be mundane, so if I want to send you 1 BTC (a unit of currency in Bitcoin) and your Bitcoin address is Z, I need to upload a script like this into the Bitcoin blockchain:

```
What's your public key? If the public key is hashed, does it equal Z?
If
yes, could you provide your private key to prove that you own this
public
key?
```

But it could be a little bit fancier. Let's say you require at least two signatures from four authorized signatures to unlock this account; you can do that with a Bitcoin script. By thinking creatively, you can produce something like this:

```
This transaction is frozen until 5 years from now. Then business will
be as
usual, that the spender must provide public key and private key.
```

However, a Bitcoin script is created with a simple programming language, so it's incapable of even looping. It's stack-based. To remedy this, you put instructions: `hash the public key`, `check a signature`, and `check the current time`. Then, it will be executed on the Bitcoin node from left to right.

This means that you cannot create a fancy program, such as an auction, on Bitcoin. Bitcoin is designed just to store and transfer value (money). So, it is purposely designed to avoid a complex program. In a Bitcoin node, every script is executed. Without a loop, a Bitcoin script is simple, and you know when it will stop. But if you have a loop in a Bitcoin script, you don't know when it will stop. It could stop in the fourth iteration, or the millionth iteration, or in a faraway future.

Some people weren't satisfied with this limitation, so Ethereum was created. The programming language that you are equipped with on the Ethereum blockchain is much more sophisticated than the programming language in Bitcoin (there is a `while` or `for` construct). Technically speaking, you could create a program that runs forever in the Ethereum blockchain.

In Bitcoin, you can store and transfer values. But there is so much more than you can do in Ethereum. As an example, you could create the following:

- A voting program
- An escrow service
- An online auction
- Another cryptocurrency on top of it

So, people like to differentiate the currencies of **Bitcoin** (**BTC**) and **Ethereum** (**ETH**). BTC is like digital gold. ETH is like oil and gas. Both are valuable if we take that analogy. But you can use oil and gas to create a whole new world, such as by creating plastics, fuel, and so on. On the other hand, what you can do with gold is quite limited, other than creating jewelry.

Creating a cryptocurrency on top of Ethereum is extremely easy. All you need is a weekend if you're a skilled programmer. You just inherit a class and set your token's name and supply limit. Then, you compile it and launch it to the Ethereum production blockchain, at which point you'd have your own cryptocurrency. Before this, creating another cryptocurrency meant forking Bitcoin. The skill level that's required to do that is quite deep (it involves C++, CMake, and replacing many parts of files in the Bitcoin core).

However, some of the complexities have been abstracted away and you don't need to know lots of low-level stuff, such as C++, most of the time.

Other types of blockchain programmers

This chapter intends to give you an intuitive understanding of how blockchain works. However, it doesn't provide a complete scope of how it works. My explanation differs quite a lot from how Bitcoin works (and even Ethereum). Ethereum does not use `SHA-256` for hashing; it commonly uses the `Keccak-256` algorithm. In our case, we only put one `history/transaction/payload` in one block, but Bitcoin can save more than 1,000 transactions in one block. Then, we generate a private key and public key by using **RSA cryptography**, while Bitcoin and Ethereum use **elliptic curve cryptography**. In our case, the payload is history (who `likes/loves/hates` an animal), but in

Bitcoin, it is a transaction that has a dependency on the previous payload. In Ethereum itself, it is a state of programs. So, if you have a variable, `a`, that's equal to an integer, `5`, in the payload, it could mean something such as "change variable `a` to integer `7`." In the Bitcoin consensus, we choose the blockchain that has the most hashing rate power, not the one that has the longest chain. For example, **blockchain A** has two blocks, but each block has the answer to solve the puzzle with 12 leading zeros, while **blockchain B** has 10 blocks but each block has the answer to solving the puzzle with only five leading zeros. In this situation, **blockchain A** has the most hash rate power.

What does it mean to be a blockchain programmer?

Let's go back to some questions. What does it mean to be a blockchain programmer? How many types of blockchain programmers are there? What is the scope of this book?

Blockchain programming could mean that you are working on improving the state of Bitcoin or creating a fork of Bitcoin, such as Bitcoin Cash. For this, you need C++ and Python. If you are creating a **Bitcoin fork**, such as Bitcoin Gold, you need to dig deeper into cryptography. In Bitcoin Gold, the developers changed the proof of work hashing function from `SHA-256` to `Equihash` because `Equihash` is ASIC-resistant. ASIC resistance means you cannot create a specific machine to do the hashing. You need a computer with a GPU to do the Equihash hashing function, but this book will not discuss that.

Furthermore, Blockchain programming could mean that you're working on improving the **Ethereum virtual machine**. For this, you need Go, C++, or Python. You need to understand how to interact with low-level cryptographic library functions. An intuitive understanding of how basic cryptography works is not enough, but this book will not discuss that either.

Blockchain programming could mean that you are writing the program on top of Ethereum. You need *Solidity* or *Vyper* for this, which this book will discuss. You only need an intuitive understanding of how basic cryptography works. Throughout this book, you will be abstracted away from low-level cryptography. Sometimes, you might need to use a hashing function in a program you write, but nothing fancy.

The scope of this book

Blockchain programming could mean that you are writing a program to interact with the program on top of Ethereum, which sounds meta. But what you will need for this depends on the platform. If it's a mobile app, you'll need *Kotlin*, *Java*, *Swift*, *Obj-C*, or even *C++*. If it's a web frontend, you will most likely need JavaScript. Only an intuitive understanding of how basic cryptography works is needed. This book will discuss some of this.

This is the same as if I asked you, what does it entail when someone wants to become a web developer? The answer is quite diverse. Should I learn *Ruby*, *Java*, *PHP*, or *Python*? Should I learn *Ruby on Rails*, *Laravel*, or *Django*?

This book is going to teach you how to build a program on top of Ethereum (not to be confused with building Ethereum itself). Comparing this with web development, this is like saying that this book is going to teach you how to build a web application using Ruby on Rails, but this book does not teach you how to dissect the Ruby on Rails framework itself. This does not mean that the internals of Ruby on Rails are not important – it just means that most of the time, you don't need them.

This book will enable you to use the Python programming language, assuming you have basic knowledge of Python already.

> **But why Python?**
> The answer is a cliché: Python is one of the easiest and most popular programming languages. It lowers the barrier to entry for someone who wants to jump into blockchain.

Summary

In this chapter, we investigated the various technologies behind cryptocurrencies, such as Bitcoin and Ethereum. This technology allows us to decentralize how we store values or code. We also covered cryptography by using private and public keys to secure the integrity of data. Later, we learned about hash functions, proof of work, consensus, and the basic concepts of blockchain programming. The blockchain will be an important technology for keeping the truth. So, now is a good time to learn about this technology.

In the next chapter, we will learn about smart contracts, which are programs that live in Ethereum. A smart contract is different than a kind of program that lives on a server, such as an application written with Ruby on Rails, Laravel, or Django. The differences are more than just the syntax; the concept is radically different than a normal web application.

References

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- `https://bitcoin.org/bitcoin.pdf`
- `https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/`

2
Smart Contract Fundamentals

In this chapter, we will explore the basics of smart contracts. While in Bitcoin we store value, in Ethereum, we can also store code. The code that we store in Ethereum is called a **smart contract**. A smart contract is a trustless piece of code, which means that the integrity of the code is guarded by algorithms and cryptography. We can store the code, which is censorship-resistant and can avoid third-party interference, even from the original developer of the smart contract. This opens up possibilities for creating many types of applications, such as transparent digital tokens, trustless crowd sales, secure voting systems, and autonomous organizations.

The following topics will be covered in this chapter:

- Flaws in traditional programs
- Smart contracts
- Secure from cheating
- Censorship resistance
- Practical applications

Technical requirements

To follow along with this chapter, you need one of the more recent versions of Linux and Python – no older than 3.8. You could try using other platforms, such as Mac or Windows, but Linux is recommended. You can also run Linux with virtualization software such as VirtualBox. We will use Ubuntu Linux 22.04 LTS and Python 3.10. The code for this chapter is available at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_2`.

Flaws in traditional programs

A smart contract is a program or an application that runs on top of Ethereum. While you can send and receive money in Bitcoin, Ethereum gives you a global computer where you can upload a program and people all over the world can use it. So, is it like a web application? No, that's not the case.

To understand the difference between traditional programs such as web applications and smart contracts, let's write a simple web application using Python:

```
$ python3.10 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install "fastapi[all]"
```

Here, we're using the `fastapi` library to build a web application quickly. Create a Python file named `main.py` and add the following code to it:

```
from fastapi import FastAPI
import os

app = FastAPI()

counter_file = "counter.txt"

counter = 0
```

With the preceding code, you import the FastAPI library and then instantiate the `app` object. After doing that, the value of the counter variable is read from the `counter.txt` file. If the file doesn't exist, you must set the `counter` variable to zero.

Then, add the following code at the bottom of the same file:

```
@app.get("/")
async def root():
    global counter
    if os.path.isfile(counter_file):
        with open(counter_file, "r") as f:
            try:
                counter = int(f.read().strip())
            except ValueError:
                counter = 0
    counter += 1
    with open(counter_file, "w") as f:
        f.write(str(counter))
    return {"counter": counter}
```

The preceding code sets the method to handle the request to the `/` path with the GET method of the `app` object that you instantiated earlier.

This code is an API server that increases a counter every time the REST endpoint is visited. The number counter is kept in a text file. However, as you can imagine, in the real world, the counter would most likely be stored in a database. This simple web application illustrates a web application that has a visitors' counter. Let's consider a web page that displays a number, such as 1,034 visitors.

To run the web application, you can run the following command:

```
(.venv) $ uvicorn main:app
INFO:     Started server process [27393]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to
quit)
```

Go to `http://127.0.0.1:8000` by using your browser or a piece of software, such as `curl`:

```
$ curl http://127.0.0.1:8000
{"counter":1}
```

Then, do it again:

```
$ curl http://127.0.0.1:8000
{"counter":2}
```

As you can see, the counter was increased by one every time you visited it. The counter is stored in the `counter.txt` file. You can check this by running the following command:

```
$ cat counter.txt
2
```

If you were to come across a web page that said "This page has been visited 100,000 times," you would likely conclude that the web page is quite popular.

But alas, people can be dishonest. Now, edit the `counter.txt` file and change the content of the file to `3000`. Access the web page again:

```
$ curl http://127.0.0.1:8000
{"counter":3001}
```

By doing this, you can see that the number that you saw on the web page could be a lie. The web page was not visited 100,000 times but the owner of the website changed the number from behind. They could manipulate the data and visitors, such as yourself, wouldn't find out.

This is not a hypothetical case. Reddit's then-CEO admitted that he edited user comments that criticized him. When you read comments from user A, it should mean the comments come from user A, not from the CEO. Would you now be able to truly trust the web application again? How can you enforce trust? In this case, because the web pages were very popular, many users could watch and notice the anomaly. Many other web pages don't have this kind of luxury.

In a normal web application, we rely on trusting the operators (developers or system administrators) of the web application. We hope that they develop and deploy the web application honestly. There is no way for us, as a user of the web application, to ensure the web application does what it claims to do.

Let's say we have a web application that hosts videos (such as YouTube or Vimeo). The web application can increase the number of likes on a video if a user clicks the **Like** button. The rule is that a user can only like a video once. Consequently, you would expect a video that contains 400 **likes** to have 400 users who have liked that video. What if I tell you that, behind the scenes, a system administrator can increase the number of **likes** artificially? This means that among 400 **likes**, it could be that only 300 **likes** come from genuine users. The other 100 **likes** are inflated by the system administrator. It does not have to be as raw as updating a table in a database directly through `UPDATE video_likes SET likes_amount = 400 WHERE video_id = 5;`. For example, the way the number of **likes** is inflated could be embedded inside the system.

Normal users wouldn't notice this. The company behind the web application could publish the source code on GitHub. But how do you make sure the web application is indeed built from the source code hosted on GitHub? What if, after deploying the web application, the system administrator or the developer patches the system?

There are some things that you can do to prevent this kind of fraudulent behavior or cheating. There are IT auditors who are employed to audit the system to make sure the system works without fraud. In the previous case, they can check the log files. Editing the user comments would leave trails. But Reddit needs to allow IT auditors to audit their system.

> **Note**
> This is why companies that handle sensitive data, such as banks, usually have their IT systems audited by a third party.

When taking this approach, you still put trust in humans, from the administrator of the system to the auditor. But what prevents auditors from being corrupted? Auditors can cheat.

With smart contracts, you place your trust in algorithms.

Smart contract

A smart contract is a different kind of program that's designed to solve the problems we encountered earlier. While in Bitcoin we store transactions, such as you sending 1 BTC to your grandma, in Ethereum, we store transactions – for example, you can change the value of a variable from 5 to 9. In Bitcoin, transactions are very narrow – they are financial. But in Ethereum, transactions are changes in the states of programs. This sounds abstract. So, let's jump into creating a program on top of Ethereum or a smart contract.

Think of a smart contract as a program or an application that lives on the blockchain. This program is in the form of a stack-based bytecode. When you interact with the smart contract, you send an instruction to the bytecode. Then, the Ethereum virtual machine will pop and push values from the bytecode and operate on the instruction according to the values from the bytecode. So, your entire program lives on the blockchain. You can imagine a smart contract as an application binary such as a `.exe` file on a Windows system, but it lives on the blockchain. In this case, you can peek into the program. This differs from a web application, where the program binary is hidden from you. You can see that the smart contract lives in each node on the blockchain, as shown in the following figure:

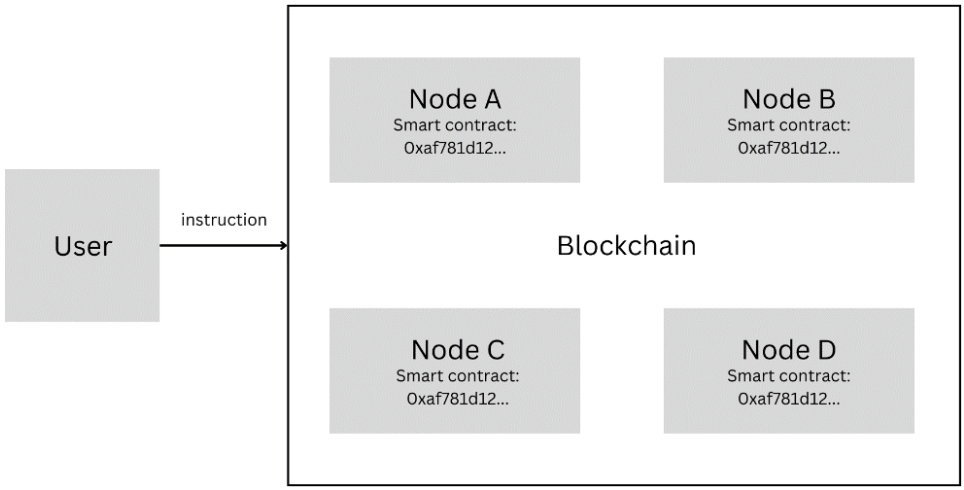


Figure 2.1: A smart contract – a program that lives on the blockchain

A user can interact with the smart contract and change the state of the program, as shown in the following figure:

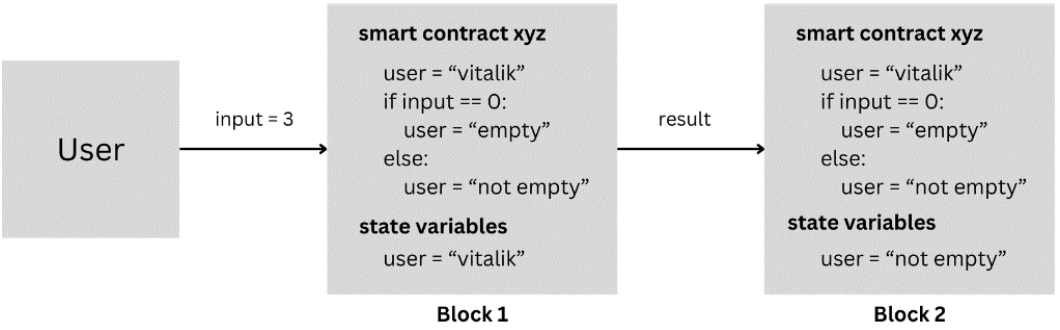


Figure 2.2: A user sends input to the smart contract

Open your browser and go to `https://remix.ethereum.org`. This will take you to the page shown in *Figure 2.3*:

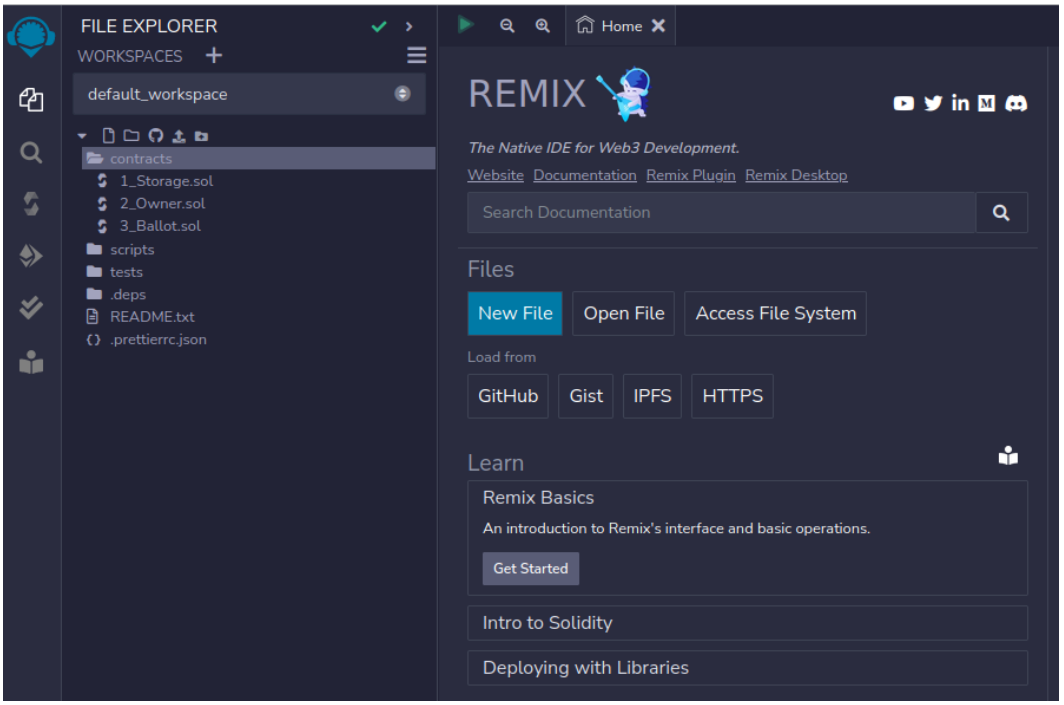


Figure 2.3: Remix

Click **contracts** under **FILE EXPLORER** and click **1_Storage.sol**. You'll see a simple smart contract:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;

/**
* @title Storage
* @dev Store & retrieve value in a variable
* @custom:dev-run-script ./scripts/deploy_with_ethers.ts
*/
contract Storage {

    uint256 number;

    /**
    * @dev Store value in variable
    * @param num value to store
    */
    function store(uint256 num) public {
        number = num;
    }

    /**
    * @dev Return value
    * @return value of 'number'
    */
    function retrieve() public view returns (uint256){
        return number;
    }
}
```

This smart contract is written using the Solidity language. There are some options to write smart contracts, just like there are choices of programming languages to write a web application.

You should understand the intention of this program, even if you never write a single line of Solidity code. Here, there is a state variable named `number`. There are two methods. The first method is called `store` and sets the value of `number`. The second method is called `retrieve` and returns the value of `number`.

Now, compile the program. To do this, you can right-click **1_Storage.sol** and choose the **Compile** menu item or you can press the green triangle button. See *Figure 2.4*:

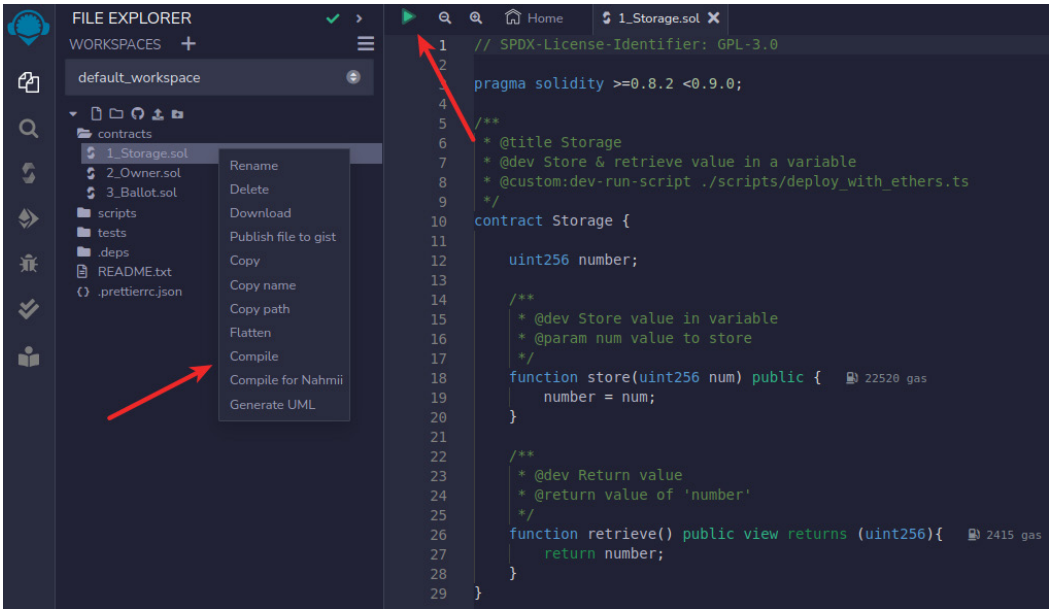


Figure 2.4: Compiling the smart contract

After compiling the program, you won't get a `.exe` file, like you would on Windows, or a `.class` file, like you would in **Java Virtual Machine** (**JVM**). Instead, you will get a `.json` file that contains the bytecode and the interface, as shown in *Figure 2.5*:

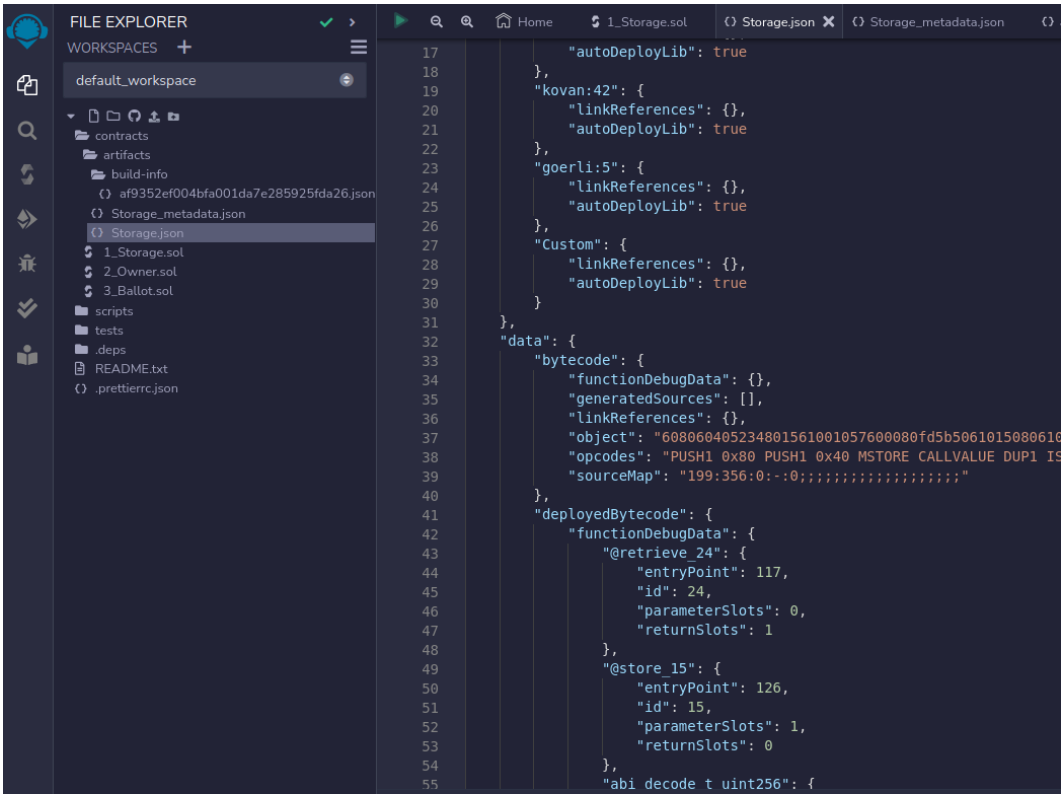


Figure 2.5: Storage.json

However, the bytecode is useless as it is. You need to upload it to the Ethereum blockchain. As an analogy, let's say you have some Python code. But to make use of it, you need to upload your Python code to a server on the internet so that people can access and benefit from your program.

For smart contracts, you need to deploy the bytecode to blockchain – or Ethereum in this case. Make sure you select **1_Storage.sol**, not **Storage.json**. Then, click the **Deploy and run transactions** menu item on the left, as shown in *Figure 2.6*:

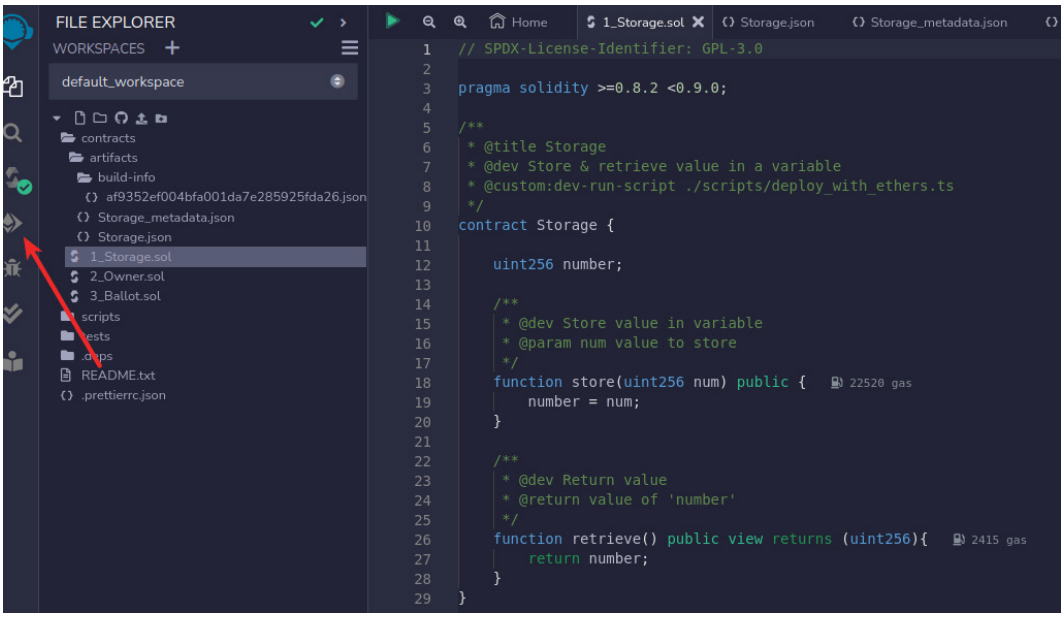


Figure 2.6: The Deploy and run transactions menu item

You'll be presented with the **Deploy** button and other options to deploy smart contracts, as shown in the following figure:

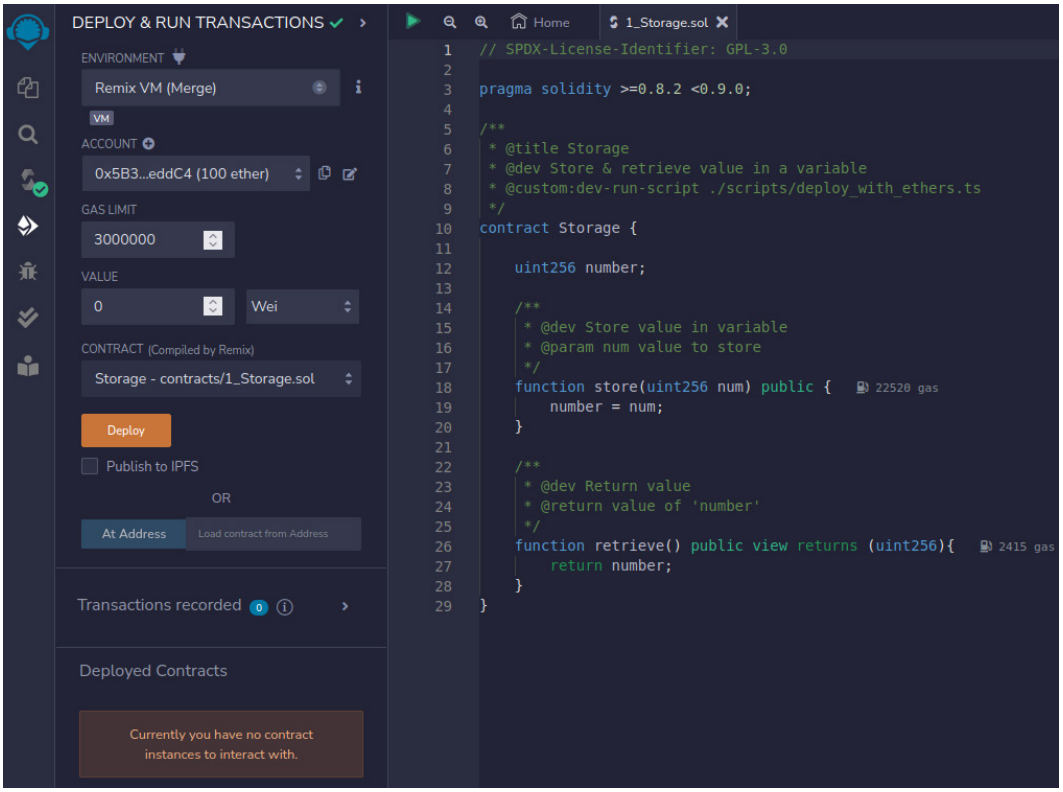


Figure 2.7: The DEPLOY & RUN TRANSACTIONS screen

We're not going to use the real Ethereum blockchain or Ethereum mainnet because that's quite expensive for learning purposes. You have to use real Ethereum currency, ETH. When people talk about Ethereum, they usually refer to Ethereum mainnet. This is where people perform transactions, such as sending ETH or deploying smart contracts. However, for development and testing purposes, there are other Ethereum blockchains, such as Ethereum Sepolia. Here, the blockchain is similar to the Ethereum mainnet, so you can conduct testing because here, ETH is not as valuable as ETH on the Ethereum mainnet. You can get ETH on Ethereum Sepolia for free. Ethereum Sepolia is not the only testing blockchain for Ethereum. There are others. Remix VM is one of them.

To deploy the smart contract, you can use the Remix VM. This Ethereum VM only lives in your browser. So, you're the only participant in this blockchain. But if you want to, you can choose to upload the smart contract to other blockchains, such as Ethereum mainnet, the one people use all over the world.

The testing blockchain provides 15 preloaded accounts with 100 ETH each. That's why it's convenient to have accounts with a lot of ETH on the testing blockchain. Don't worry about the **Gas Limit** and **Value** fields. Make sure that, in the **Contract** field, you have a value of **Storage – contracts/1_Storage.sol**.

Now, click the **Deploy** button to upload your bytecode to the blockchain that lives in your browser.

Once you've done this, you'll notice that the account has less than 100 ETH. To deploy a smart contract, you need to spend ETH. How much? It depends on whether there's congestion on the Ethereum network and the complexity of the smart contract. In this testing blockchain, it's cheap. See *Figure 2.8*:

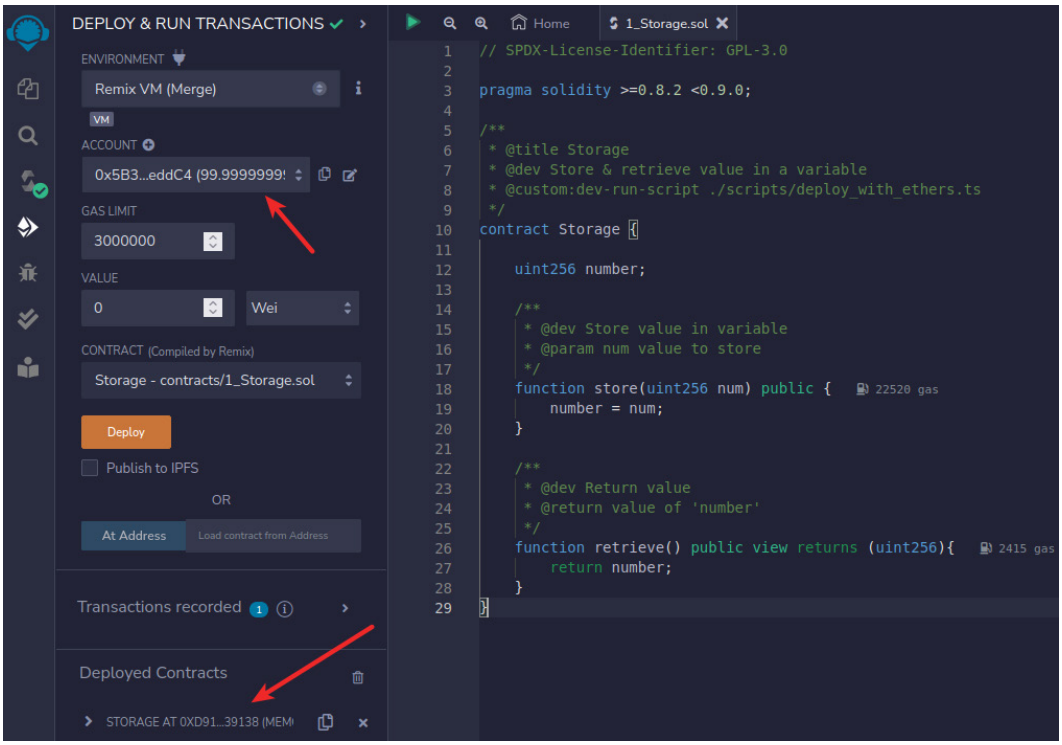


Figure 2.8: The deployed contracts

Also, notice that there is an address in the **Deployed Contracts** field. This is where your smart contract lives. This address is like the URL of a web application. Without this URL, people won't be able to find where your web application is. Similarly, without the address (on the blockchain), people will be unable to find out where your smart contract is.

After deploying the smart contract, it's time to interact with it. Expand the address field below the **Deployed Contracts** menu item; you will see options to execute the methods of the smart contract, as shown here:

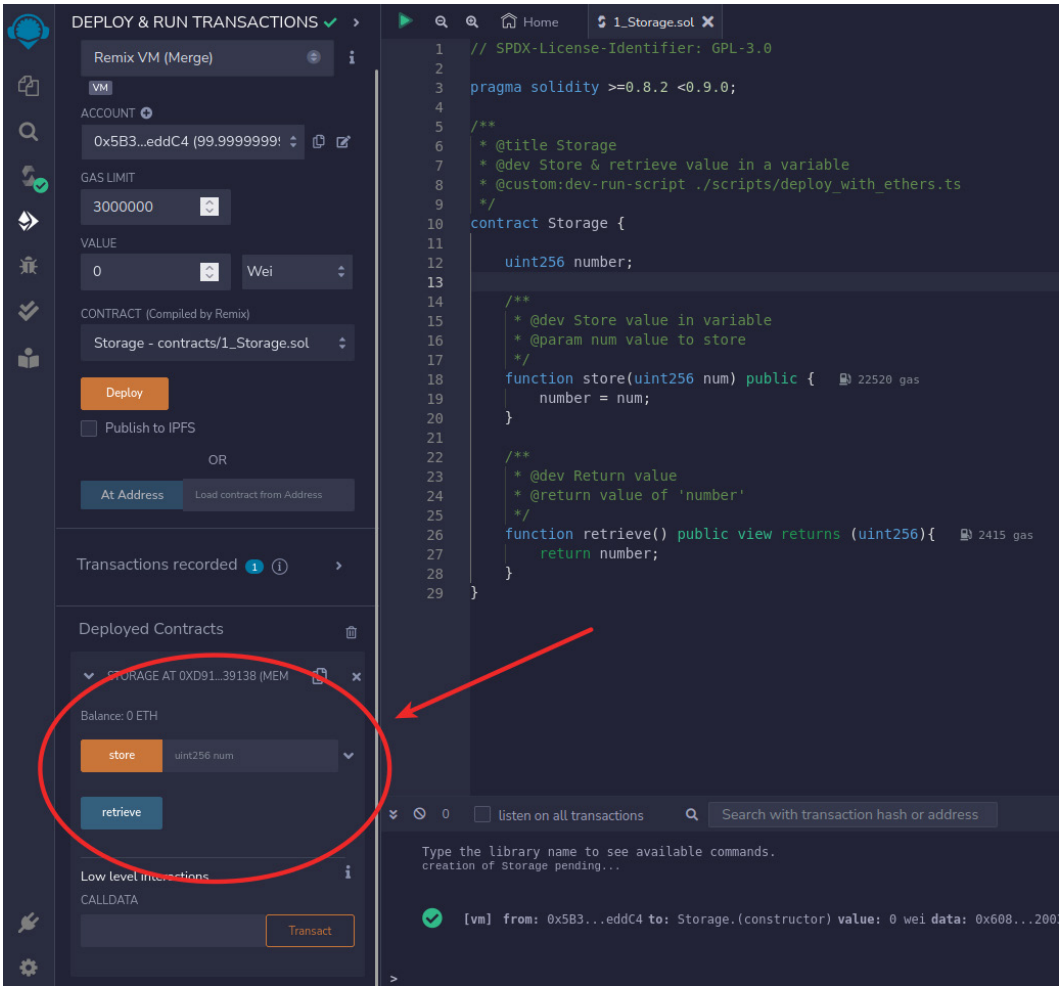


Figure 2.9: Fields to interact with the smart contract

As you can see, there are two fields representing the two methods of our smart contracts. You can ignore the **Low level interactions** field below them.

Let's access the `store` method of the deploy smart contract. Put `value 6` into the text field beside the orange **store** button, then press that button.

The **retrieve** button doesn't have the text field on the right-hand side because the `retrieve` method in the smart contract doesn't accept any parameter. Click the **retrieve** button.

You will see that a string appears below the button: `0: uint256: 6`. Here, `0:` means the first element of the returned result. The method of the smart contract can return many elements. In our case, it's only one. `uint256` specifies the data type of the returned `value` method, which is `6`.

If you want to, you can set another value with the `store` method. Then, click the **retrieve** button to get the value; it will be different.

So, what are the differences compared to a normal web application? Web applications can also store a value and retrieve it later.

One of the differences is that the state changes in a web application are not transparent. We've discussed this earlier. The state changes in a smart contract are as clear as water.

As you may have noticed, there are transaction logs in Remix:

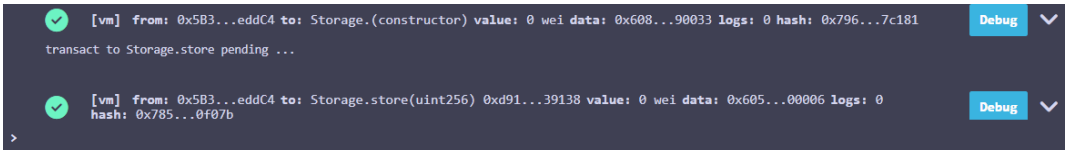


Figure 2.10: Transaction logs

All transactions will be logged or recorded on the blockchain. The first transaction was recorded when you deployed the smart contract. The second transaction was recorded when you accessed the `store` method with `value 6`. Let's click that transaction log – the second line with the green check mark. See *Figure 2.11*:

Figure 2.11: The decoded input

If you expand the second transaction log, you will see your input, `value 6`, in the decoded input field. If you were to deploy the smart contract to the Ethereum mainnet and change the value of `number` with the `store` method, the transaction would be recorded on the blockchain. If you were to deploy the smart contract to the Ethereum mainnet, this information would be available to people all over the world!

Now that we've covered a toy example, let's discuss the security property of smart contracts!

Secure from cheating

To emphasize the security property (from cheating) of a smart contract, you can visualize the states changing, as shown in *Figure 2.12*:

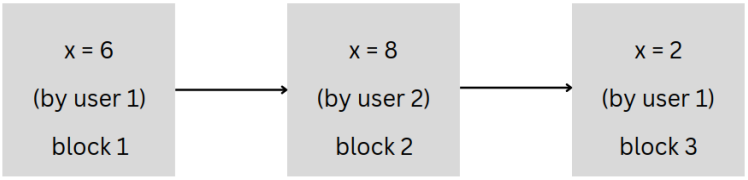


Figure 2.12: The state of the x variable changing

Every time someone changes the value of `x`, the transaction will be recorded on the blockchain. User 1 stores a value of `6` in `x` and the information is recorded on the first block. Then, user 2 stores a value of `8` in `x`, and the information is recorded on the second block. Lastly, user 1 stores a value of `2` in `x`, and the information is recorded on the third block.

Of course, in the real world, the state changing is not conveniently put in sequential blocks. User 1 changes the value of `x`, which is recorded on block 3, and then nothing happens for a long time. User 2 then changes the value of `x` in block `1000`.

The important thing to note here is that the information that we get is not from only one smart contract. Let's say three smart contracts with similar code are changing the value of `x` and retrieving it. These changing states can be seen in *Figure 2.13*:

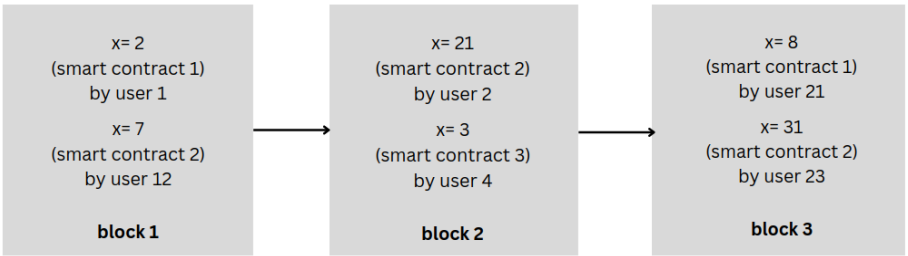


Figure 2.13: The state changing for many smart contracts

Also, in a block, other information is recorded as well, such as the timestamp of the block and which validator validates this block and proposes it to the Ethereum network.

Now that you understand one of the unique properties of smart contracts, let's get back to the Reddit cheating problem.

As an analogy, if Reddit is built on the blockchain and the then-CEO of Reddit changes the message of a user, it will be caught very soon and the proof will be visible on the blockchain. People will be able to point to a specific block where the then-CEO of Reddit changed the message of the user. The then-CEO of Reddit would have been unable to hide the cheating because the blockchain system prevents it. In the original case, many users did notice that the message had changed. However, many such cases of cheating in the real world don't get exposed because not everyone has time to notice the problems.

> **Note**
>
> With blockchain, cheating of this kind can be prevented from the get-go. The blockchain has been designed in such a way that it prevents cheating from the beginning.

Having said that, Reddit cannot be built on Ethereum with default settings because storing a lot of messages on the blockchain is expensive.

You can take a peek at information stored on the Ethereum mainnet on Etherscan. For example, you can take a look at the information on block 17249745 (`https://etherscan.io/block/17249745`). There are 112 transactions and 37 contract internal transactions. One of the 112 transactions, which involves transferring 4,000 USDC from one address to another, can be seen at `https://packt.link/QokNP`:

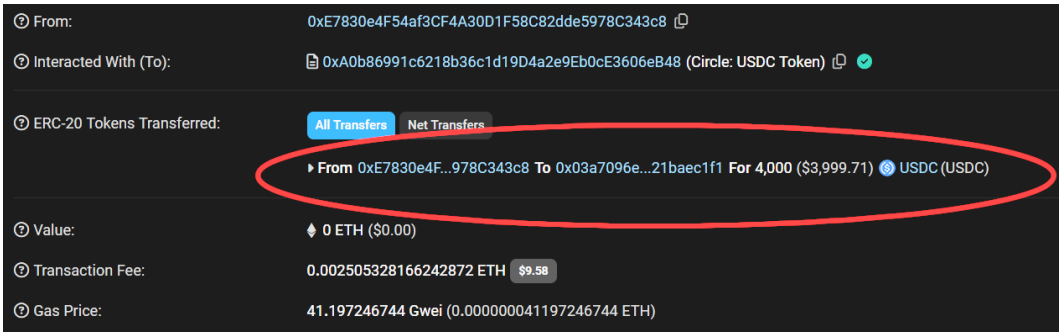


Figure 2.14: The USDC transfer transaction, as seen via Etherscan

So, nothing can be hidden from people on the blockchain. Everything is recorded on the blockchain.

Other than all transactions being recorded on the blockchain, there is another side to transparency when it comes to smart contracts, which is the **transparency of the program**. In a web application, you don't get access to the source code. You don't know what will happen if you send an input to the web application. The deployer of the web application can provide the source code on GitHub and declare that that is the source code of the web application. However, that is not a guarantee that the source code is being used in production.

But you can make sure the source code of the program on the blockchain is being used by the program on the blockchain. You can verify this yourself. To do this, you must get the bytecode of the smart contract from its address on the blockchain. Then, you must compile the source code of the smart contract. The result of this compilation is the bytecode. Then, you can compare both bytecodes; if the results are the same, the smart contract on the blockchain is verified. Here's how it looks when done with code:

```
from web3 import Web3
w3 = Web3(Web3.HTTPProvider('https://mainnet.infura.io/v3/YOUR_INFURA_
PROJECT_ID'))
bytecode = "0xafaf0707..." # bytecode from the compilation of the
smart contract's source code
contract_address = "0xffaabb..."
deployed_bytecode = w3.eth.get_code(contract_address).hex()
if bytecode == deployed_bytecode:
    print("Smart contract verified.")
else:
    print("Smart contract verification failed.")
```

People have the option to upload the smart contract's source code to Etherscan so that they can check whether the smart contract at a certain address comes from the source code that people have uploaded.

This opens up a lot of possibilities. It means people can audit a program and decide for themselves whether they want to use the program or participate in the application as users.

Let's assume that you're considering participating in a token smart contract, but you notice that the deployer of the smart contract can decrease and increase the balance of any account in the smart contract on their whim. This is undoubtedly not good. Due to this, you decided not to use a smart contract.

There is another kind of smart contract in which you may find that the rules are more fair. No one can increase or decrease the balance of any account without undertaking the transferring token method. This, I believe, is fair.

A smart contract is a truly "open source" program!

Censorship resistance

Another property of smart contracts is censorship resistance. This means that smart contracts, unlike web applications, are much harder to censor. Your web application is hosted on the cloud, but your account can be shut down on a whim. A web application also can be taken down either by requests from third parties or by the admin of the program.

But the programs on the blockchain "live" forever. So long as the Ethereum mainnet is up, people can interact with the programs. As an analogy, it's like your web application lives forever until the internet as a whole is shut down.

This means that you can create a program where even you cannot censor any users who interact with your program on the blockchain. You can write a program such that no superuser can block or manipulate the balances of something in the smart contract. It becomes harder to censor your smart contract if there is no superuser.

> **Note**
>
> To censor web applications, you can block their URLs – for example, to block Facebook in a country, you can block the `https://facebook.com` URL. However, technically, it's impossible to censor addresses on the blockchain. To censor smart contracts, you need to shut down or block the whole Ethereum mainnet.
>
> If not, you need to use other legal means, such as threatening people not to interact with certain addresses on the blockchain. If you use this smart contract, you violate the law and might end up in jail. That's the best they can do. Censoring smart contracts is much harder than censoring web applications.

One example of this would be when the US treasury department blacklisted Tornado Cash, a cryptocurrency mixing service, in August 2022. Although the US government prohibited its people from using this smart contract, people living in the US can technically continue to use it. However, the transactions can be traced to some extent. The US cannot delete the smart contract and it's impossible to shut down the whole Ethereum blockchain. But in the case of web applications, the US has shut down many.

Why is this useful? Not everyone or every country is happy with the US being the world police. Having applications that are hard to be censored by the US is beneficial to some countries. For example, if a person wants to donate a sum of money to another person in a different country via PayPal, such a donation can be blocked by the US company. Remember that PayPal is a US company and must follow US regulations, something that might not be beneficial for some countries in the world.

To censor blockchain applications, the US is working with some startups that analyze transactions on the blockchain. If they are confident that an address is tied to North Korean hackers, scammers, or drug traffickers, for example, they will put the address into a list. Then, they will send the list to cryptocurrency exchanges to make sure that the exchanges will block these addresses from using their service.

In a way, you can say that blockchain applications or smart contracts can be censored (by the US) but that it gives other countries a platform that cannot be censored by the US.

Practical applications

Storing data in the blockchain is expensive. However, there is a workaround, something you'll learn later. For now, just assume you cannot store a lot of data on the blockchain. This means that you cannot build Reddit-like applications. So, what kinds of applications you can build on the blockchain? An application's most important value is security from cheating. However, the data that's stored shouldn't be huge. What kinds of applications cannot store a lot of data but need to make sure the data cannot be tampered with easily?

Financial applications! The popular term for this is **decentralized finance** (**DeFI**). What we mean by financial applications are those that involve currencies, lending applications, exchanges between cryptocurrencies, futures, auctions, crowdfunding, and many more. They don't need to store a lot of messages because transferring money means increasing the balance of an account and decreasing the balance of another account. Such data is miniscule.

The security from cheating property is paramount. Changing a message from "There are many people!" to "There are few people!" is one thing. What we have here is merely a fake message. However, manipulating an account's balance so that it goes from $100,000 to $10,000 can be fatal to a user. For most people, losing money has more destructive implications than being slandered.

The following are some applications you can find on the blockchain that are being used every day:

- **Uniswap**, a decentralized exchange between cryptocurrencies
- **Compound Finance**, a lending application
- **Juicebox**, a fundraising and organization management
- **Safe**, a collective asset management platform or a bank
- **Yearn**, a hedge fund application

With that, let's summarize this chapter.

Summary

In this chapter, we saw that traditional applications, such as web applications, are prone to cheating. We deployed a simple smart contract to experiment with it on our browser. By doing so, we learned that blockchain is a platform that's designed to be safe from cheating for various applications. Then, we became aware that smart applications are technically harder to censor than web applications. In the next chapter, we'll learn how to write smart contracts with Vyper, a programming language similar to Python.

3

Using Vyper to Implement a Smart Contract

This chapter will cover the process of writing smart contracts using Vyper, a programming language that bears a resemblance to Python. In this chapter, you'll learn how to set up the Vyper compiler and deploy the smart contract on the blockchain. Then, we'll go through an overview of Vyper by writing small chunks of code to understand one feature at a time. In the previous chapter, you wrote a simple smart contract using Solidity. Solidity is, indeed, the most used programming language used to write smart contracts. But for Python developers, Vyper is more familiar.

The following topics will be covered in this chapter:

- Setting up Vyper
- Data types
- Functions
- Control structures
- Environment variables
- Event logging
- Interface

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_3`.

Setting up Vyper

There are a couple of programming languages used to write smart contracts. The most popular is Solidity. But the syntax of Solidity is similar to JavaScript, and that's a bit of a turn-off for some people, especially Python developers. Vyper is similar to Python, and Python developers can be more productive with Vyper. Vyper has been used in some big projects, such as Curve Finance, the **decentralized exchange** (**DEX**) and **automated market maker** (**AMM**) for stablecoins.

Vyper was created by Vitalik Buterin, the creator of Ethereum. Vitalik also wrote Serpent, the predecessor to Vyper, in 2014. Then, he built Vyper in 2016. He does not work on it anymore. His latest commit was in late 2017. Now, Vyper is managed by a group of dedicated maintainers.

The way to install Vyper depends on your operating system. On Ubuntu Linux, make sure you have Python with a minimum version of 3.8 installed. Also, make sure you have a compiler or Python development files installed. To install Vyper, it's recommended to use a Python virtual environment. The following commands will install Vyper inside a Python virtual environment:

```
$ mkdir vyper_tutorial
$ cd vyper_tutorial
$ python3.10 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install vyper
```

Using the `pip` installer, you can isolate the Vyper programming language in a directory. To test whether Vyper has been correctly installed, you can run Vyper like this:

```
(.venv) $ vyper --version
0.3.10+commit.9136169
```

At the time of writing, the latest version of Vyper is 0.4.0b6 and the latest stable version is 0.3.10. However, you might read this book in the future when the version of the Vyper has increased. The later version of Vyper might have different features, bugs, or behaviors. Therefore, to make sure you don't encounter difficulties in following this tutorial, it's recommended to install a specific earlier version of Vyper:

```
(.venv) $ pip install vyper==0.3.10
```

When you want to work on real projects, you can always use the latest version of Vyper.

You can also use Docker if you don't want to use the Python virtual environment:

```
$ docker pull vyperlang/vyper
$ docker run vyperlang/vyper --version
0.4.0b7+commit.b43fface
```

The latest version of Vyper from Docker is 0.4.0b7. However, we want to use Vyper 0.3 in this book. Either the 0.3.9 or 0.3.10 version is fine, but we'll use 0.3.10 in this chapter. To install and use a specific version of Vyper with Docker, you can run this command:

```
$ docker pull vyperlang/vyper:0.3.9
$ docker run vyperlang/vyper:0.3.9 --version
0.3.9+commit.66b96705
```

Now, to get the hang of Vyper, let's compile a simple Vyper file. First, create a Vyper file and name it `Storage.vy`. The extension for Vyper files is `.vy`. Add the following code to the file:

```
# @version ^0.3.0

number: uint256

@external
def store(num: uint256):
    self.number = num

@external
def retrieve() -> uint256:
    return self.number
```

This is the application that you worked on in the previous chapter – a smart contract that can store a value in a variable and retrieve it. By looking at it at a glance, you can see the code is similar to Python. The first line is a version pragma:

```
# @version ^0.3.0
```

This means you can only compile this Vyper file with the Vyper compiler from at least version 0.3.0.

Then, you will see a variable declaration. *The variable has a name "number" and the data type of it is uint256.*

Finally, you will have two functions' declarations. Each of them gets an annotation.

To compile this Vyper file, run the following command:

```
(.venv) $ vyper Storage.vy
0x61005d61000f60003961005d6000f36003361161000c57610048565b5f3560e01c3
461004c57636057361d81186100305760243610610004c576004355f55005b632e64ce
c18118610046575f5460405260206040f35b505b5f5ffd5b5f80fda16576797065728
3000309000b
```

The output is the bytecode of the smart contract. However, most of the time, you will want the interface or **Application Binary Interface** (**ABI**) of the smart contract as well. The reason is that the ABI makes it possible for users to interact with the smart contract. The ABI tells the user what functions are available for users and how to execute functions. So, you should compile the file with this option:

```
(.venv) $ vyper -f abi,bytecode Storage.vy
[{"stateMutability": "nonpayable", "type": "function", "name":
"store",
 "inputs": [{"name": "num", "type": "uint256"}], "outputs": []},
{"stateMutability": "nonpayable", "type": "function", "name":
"retrieve", "inputs": [], "outputs": [{"name": "", "type":
"uint256"}]}]
0x61005d61000f60003961005d6000f36003361161000c57610048565b5f3560e01c34
61004c57636057361d81186100305760243610610004c576004355f55005b632e64cec1
8118610046575f5460405260206040f35b505b5f5ffd5b5f80fda16576797065728300
0309000b
```

The first line of the output is the ABI of the smart contract. It tells a user how to interact with this smart contract. The second line of the output is the bytecode. This smart contract behaves identically with the smart contract you interacted with in the previous chapter.

If you want to compile Vyper files with Vyper from Docker, you can run the following command:

```
$ docker run -v $(pwd):/code vyperlang/vyper:0.3.10 -f abi,bytecode
/code/Storage.vy
[{"stateMutability": "nonpayable", "type": "function", "name":
"store", "inputs": [{"name": "num", "type": "uint256"}], "outputs":
[]}, {"stateMutability": "nonpayable", "type": "function", "name":
"retrieve", "inputs": [], "outputs": [{"name": "", "type":
"uint256"}]}]
0x61005d61000f60003961005d6000f36003361161000c57610048565b5f3560e01
c3461004c57636057361d8118610030576024361061004c576004355f55005b632e
64cec18118610046575f5460405260206040f35b505b5f5ffd5b5f80fda16576797
065728300030a000b
```

You used a specific version of Vyper image with Docker in the compilation process. Otherwise, the complication would fail because we required the Vyper file to be compiled with the Vyper compiler with version 0.3.

You can also compile Vyper files using Remix, just like you did in *Chapter 2*. Open `http://remix.ethereum.org` and connect your Vyper compiler to Remix. There is a Vyper compiler web server tool that you can run, called the **Hosted Compiler**, which you can get from this URL: `https://github.com/ApeWorX/hosted-compiler`. To run and install the tool, follow these steps:

```
$ git clone https://github.com/ApeWorX/hosted-compiler
$ cd hosted-compiler
$ python3 -m venv .hosted-compiler-venv
$ source .hosted-compiler-venv/bin/activate
```

```
(.hosted-compiler-venv) $ pip install uvicorn fastapi eth-ape'
[recommended-plugins]'
```

Then, you can run the tool with this command:

```
(.hosted-compiler-venv) $ uvicorn main:app
INFO:     Started server process [74783]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to
quit)
```

Alternatively, you can use Docker to run the Hosted Compiler tool:

```
$ docker run -p 8000:8000 ghcr.io/apeworx/hosted-compiler:latest
```

You can compile Vyper files by sending the content of the files to the web server that listens to port `8000`.

Then, you must install the Vyper plugin on Remix. Click the **PLUGIN MANAGER** menu item on the left. Then, find the Vyper plugin.

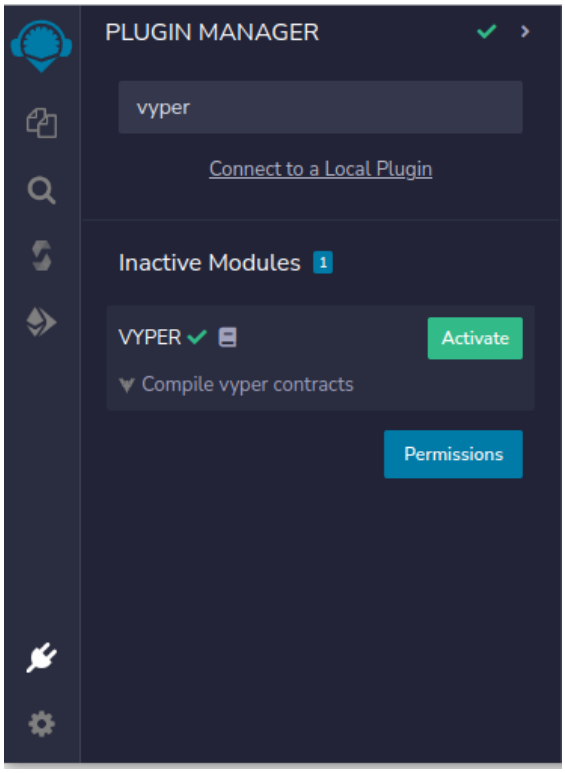


Figure 3.1: The Plugin Manager

Remix has many plugins; one of them is Vyper. Click **Activate**.

Then, a new menu item on the left side will appear, called **VYPER**. Select it. Then, you will see the following screen:

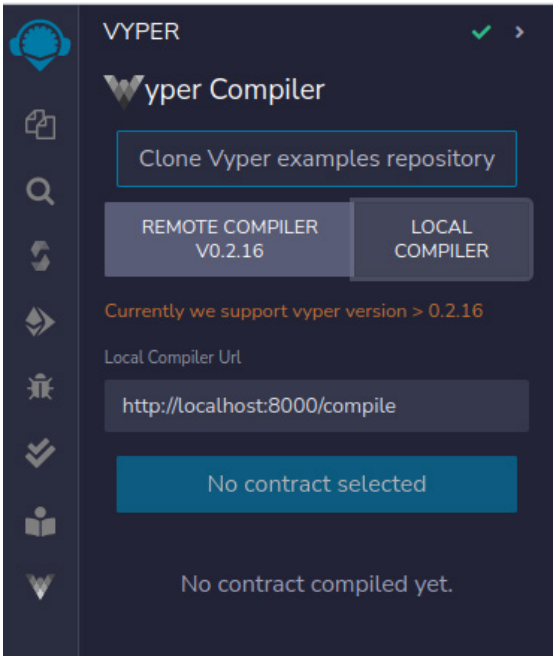


Figure 3.2: The Vyper compiler

Remix can use the Vyper compiler. You can use the remote compiler, but in this case, you would need to use the local compiler.

Choose **Local Compiler**, and make sure the URL is `http://localhost:8000/compile` in the **Local Compiler Url** field. This is where the Hosted Compiler tool ran previously.

Now, click the **FILE EXPLORER** menu item. Then, inside the `contracts` folder, upload `Storage.vy` (see *Figure 3.3*).

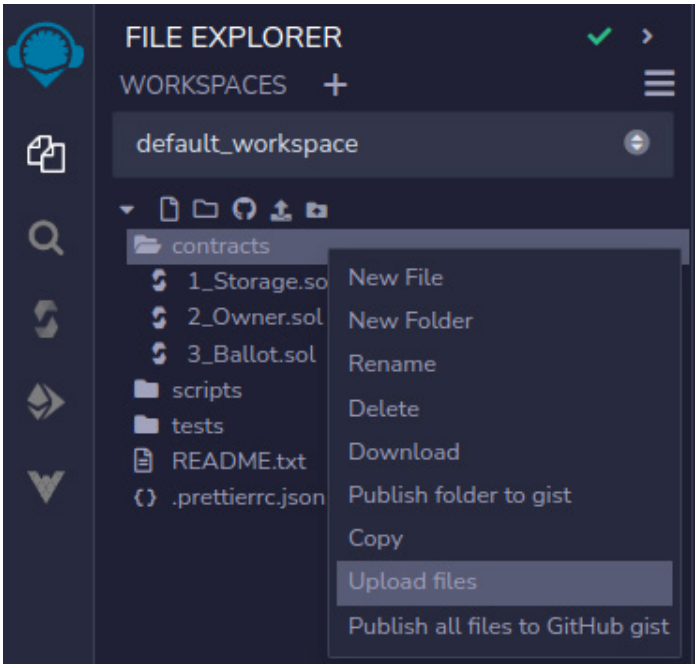


Figure 3.3: File Explorer

This is the **FILE EXPLORER** tab. You want to put the Vyper files inside the `contracts` directory alongside `1_Storage.sol`, `2_Owner.sol`, and `3_Ballot.sol`. Make sure the `contracts` directory is selected by right-clicking it.

Then, make sure `Storage.vy` is selected before going back to the **Vyper Compiler** menu item. Now, you have an option to compile `contracts/Storage.vy`.

But before you compile the file, remove this line from `Storage.vy`:

```
# @version ^0.3.0
```

There is a bug in the Hosted Compiler where it cannot handle the pragma version line.

Click the button to compile the Vyper file. Then, you'll have the result in ABI, bytecode, runtime bytecode, and LLL, as shown on the following screen:

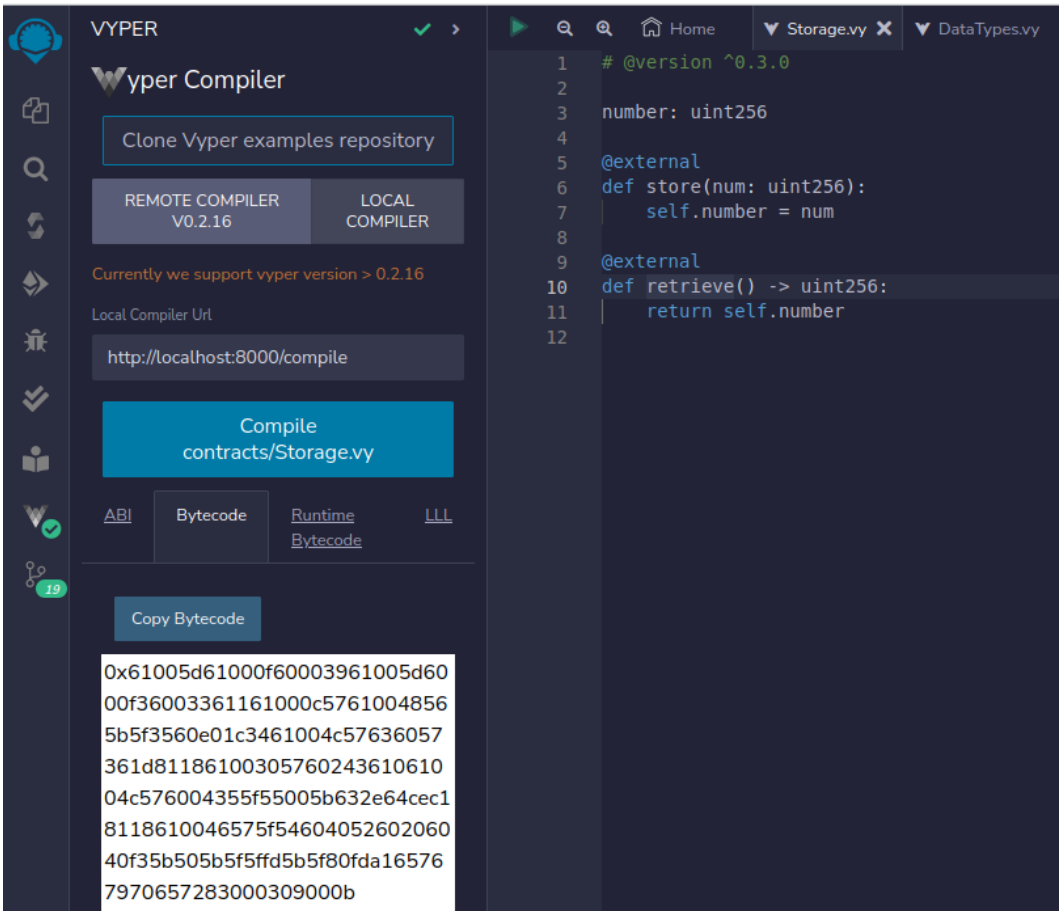


Figure 3.4: The compilation result

The **Vyper Compiler** tab gives you the results in tabs. If you want to get the bytecode, choose the **Bytecode** tab.

To deploy the bytecode, click the **DEPLOY & RUN TRANSACTIONS** menu item. Then, you will have an option to deploy the `Storage` smart contract. Click the **Deploy** button (see *Figure 3.5*).

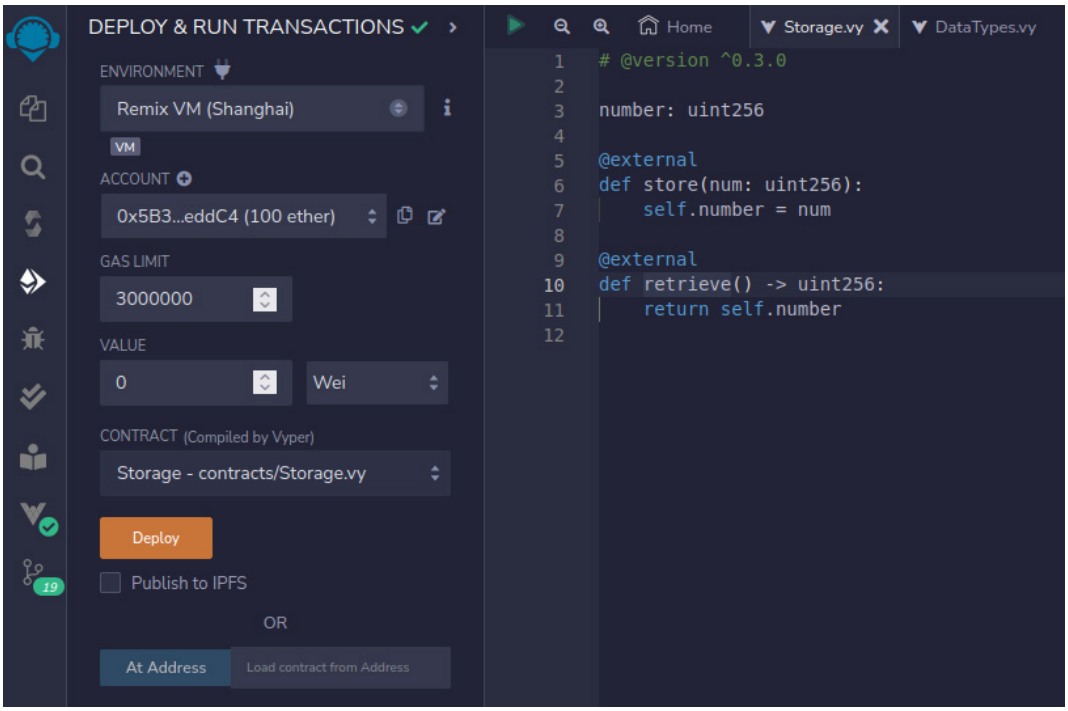


Figure 3.5: Deploying the Storage smart contract

This page should look familiar to you. Make sure the **CONTRACT** field shows `Storage – contracts/Storage.vy` when you deploy the smart contract. Once this is done, you can interact with the smart contract, just like you interacted with the smart contract in *Chapter 2*.

In the next sections, we will focus only on compiling Vyper files and not on running Vyper smart contracts. So, we will compile Vyper files in the command line instead of using Remix. However, you're free to compile Vyper files with Remix, and you can interact with the smart contracts afterward.

We start by learning what data types are available in Vyper. This is crucial because, on the blockchain, storage is expensive. Hence, it is important that you understand the trade-offs between data types.

Data types

After the version pragma, you can declare variables. The variable can have different data types. Create a file named `DataTypes.vy`, and then add some data types to it:

```
# @version ^0.3.0

life_is_beautiful: bool
var_int1: int8
var_int2: int64
var_int3: int128
var_int4: int256
var_uint1: uint8
var_uint2: uint64
var_uint3: uint128
var_uint4: uint256
pi: decimal
```

If you come from languages such as C/C++, you should be familiar with these data types. However, Vyper has more data types. Add the following data types below the previous code:

```
my_grandma_wallet: address
var_byte1: bytes32
var_byte2: bytes18
var_bytes: Bytes[56]
author: String[100]
enum Direction:
    NORTH
    SOUTH
    WEST
    EAST
direction: Direction
my_list: int16[5]
my_dynamic_array: DynArray[int128, 5]
struct Permission:
    write: bool
    execute: bool
my_permission: Permission
donaturs: HashMap[address, uint256]
```

You can compile the file by using this command:

```
(.venv) $ vyper -f abi,bytecode DataTypes.vy
[]
0x61000d61000f60003961000d6000f3a165767970657283000309000b
```

Note that the ABI is empty because you didn't define any function to interact with the smart contract. It only contains variable declarations.

In this Vyper file, you declared many variables with so many different data types. Let's look into some of these data types.

Boolean

The first variable declaration is a Boolean-type variable. The Boolean data type represents two possible values – `True` or `False`:

```
life_is_beautiful: bool
```

The Boolean variable is useful for deciding the control flow of the smart contract.

Signed integer

A smart contract like a DeFi application needs to hold money. To represent the money, you can use this data type. The following lines are signed integer-type variables:

```
var_int1: int8
var_int2: int64
var_int3: int128
var_int4: int256
```

A signed integer means negative or positive integers. The variable can hold a negative integer such as -2 or a positive integer such as 5. It cannot hold decimal values such as 5.4.

The 8 in `int8` or 64 in `int64` means the bit or the size of the integer data type. The variable with the `int8` data type can hold the value from -128 until 127. We get 128 from 2 to the power of 8. The 127 comes from 2 power to 8 minus 1. It cannot hold the value of 1,000. To store 1,000, you can use the `int16` data type or a bigger one, such as `int256`.

Unsigned integer

This data type is similar to a signed integer. The following lines are unsigned integer variable declarations:

```
var_uint1: uint8
var_uint2: uint64
var_uint3: uint128
var_uint4: uint256
```

Unlike a signed integer, a variable with an unsigned integer data type cannot store the negative value. The 8 in `uint8` means the same thing. A variable with the `uint8` data type can only store a value from 0 to 127.

Decimal

Sometimes, you need to store a number that has fractions – for example, $4.30 (4 dollars and 30 cents). Here, you can use the decimal data type. The following line here is a decimal variable declaration:

```
pi: decimal
```

A variable with the decimal data type can store decimal values, such as 3.14 or -7.

Address

If you've written many Python and C programs, you must be familiar with the Boolean, unsigned integer, signed integer, and decimal data types.

The address data type, however, is a unique data type that you don't find in other programming languages.

The example of an address variable's value is `0xde93510CFa39Ab92BF927399F799DbE71997Ee0b`. The address data type is a 20-byte value. The address must be written in hexadecimal form with the leading `0x`. So, the length of the address data type value is 42. One byte takes two characters, and there are 20 bytes. The leading `0x` takes two characters.

The address data type variable refers to the address on Ethereum. The address is where your Ethereum or smart contract lives. If you want to send some ETH to your grandma, she must give her your Ethereum address, which can be stored inside this address variable.

Fixed-size byte array

A fixed-size byte array is a more general form of the address data type because you can store bytes with different sizes. The following lines are fixed-size byte array variable declarations:

```
var_byte1: bytes32
var_byte2: bytes18
```

You can store byte values in these data type variables, as many as the fixed size of the data type. A byte value is something like `0xabcf07af`. Yes, it's similar to the address data type value. In fact, the address data type is almost the same as `bytes20`. The 32 in `bytes32` or 18 in `bytes18` refers to the size of the bytes the variable can hold. You cannot store 32 bytes in a variable with the `bytes18` data type. There are 32 choices for you, from `bytes1` to `bytes32`. No, there is no `byte33` or `byte1000` data type.

Byte array

The byte array data type is similar to the fixed-size byte array, but it's more flexible. The following line is the byte array variable declaration:

```
var_bytes: Bytes[56]
```

If 32 bytes is not enough, then you can use a byte array that can hold a bigger number of bytes. You can put a bigger number in the bracket. `56` in `Bytes[56]` indicates the maximum size of bytes that you can store in the variable.

String

What if you want to store characters? You can use the string data type. The following line belongs to a string data type:

```
author: String[100]
```

In Vyper, you need to declare the maximum size of characters that this `String` variable can hold.

Enum

There will be times when you would want to limit values in a variable. For example, you want to keep the direction value in a variable. In other words, you only want to store one of four values, such as `NORTH`, `SOUTH`, `WEST`, and `EAST`, in the variable. So, you can use `enum`:

```
enum Direction:
    NORTH
    SOUTH
    WEST
    EAST
```

This is the data type, not a variable declaration. To declare a variable with the `Direction` `enum` type, you have to do it differently:

```
direction: Direction
```

The variable can only accept one of four values declared in the `Direction` enum.

List

What if you want to create an array or a list of integers? You can use a list syntax like this:

```
my_list: int16[5]
```

This means this list can hold `5` variables with the `int16` data type.

Dynamic arrays

What if you're not sure how big the list you're going to need is? You can use a dynamic array – `my_dynamic_array: DynArray[int128, 5]`.

The size of the `my_dynamic_array` variable is not 5 but 0 because there is no value stored in this variable yet. However, later, you can insert a value into this variable, and then the size becomes 1. If you insert another value into this variable, the size becomes 2. So, we can infer that the size is dynamic. But you cannot add more than five values to this variable. The `5` in `DynArray[int128, 5]` indicates the maximum size that this dynamic array can expand to.

Struct

You may reach a stage when you want to convert multiple data types into one data type. The following line is the struct data type that can achieve that:

```
struct Permission:
    write: bool
    execute: bool
```

This is the declaration of the `Permission` struct data type. The variable declaration happens in the next line:

```
my_permission: Permission
```

The `my_permission` variable can hold two `bool` values combined in a struct.

Mapping

The mapping data type is similar to a hash or a dictionary. A mapping consists of keys and values. The key points to the value. The declaration of a mapping variable takes place in the last line of the Vyper file:

```
donaturs: HashMap[address, uint256]
```

The variable holds the relationship between the `address` keys and `uint256` values. You can think of the `donaturs` variable as a mapping that associates each Ethereum address (the key) with the corresponding donation amount in `uint256` format (the value).

Right now, all of these variables have default values because you haven't initialized them yet. The default values are `False` for boolean, `0` for integer types, an empty string for `String`, zero bytes for byte data types, and `0x0000000000000000000000000000000000000000` for the address data type.

Until now, you have only declared variables with data types. The variables that you've declared are called state variables, and they define the state of the smart contract. However, you haven't initialized them. To initialize them, you need to do it inside functions. Initializing variables means giving other values to the variables other than default values. For example, you may want to set a variable with the `uint128` data type, with the value `5`.

Functions

A function in Vyper is exactly like a function in a Python program. You group statements inside a code block unit and execute them anytime you want. The code block is the function. You've seen functions in `Storage.vy`. There are two functions in that file, the **store function** and the **retrieve function**.

As you did with the store function, you can create a function to set the values for the variables that you've declared. Then, you can call the function after deploying the smart contract to the blockchain. However, this brings us to the question, wouldn't it be better if there was a function that would be executed automatically when the smart contract is deployed? There is a special function, called `__init__`, just like in Python!

Create `StorageInit.vy` and add the following code to it:

```
# @version ^0.3.0

my_grandma_wallet: address
author: String[100]
fib_list: int16[5]
fib_dynamic_array: DynArray[int128, 5]
struct Permission:
    write: bool
    execute: bool
my_permission: Permission
donaturs: HashMap[address, uint256]
```

These are the state variable declarations that you've done in the previous section. Now, let's add a function to initialize them. Add the following code after the code you've added:

```


@external
def __init__():
    self.my_grandma_wallet =
0xde93510CFa39Ab92BF927399F799DbE71997Ee0b
```

```
    self.author = "Arjuna Sky Kok"
    self.fib_list = [1, 1, 2, 3, 5]
    self.fib_dynamic_array.append(1)
    self.fib_dynamic_array.append(1)
    self.fib_dynamic_array.append(2)
    self.my_permission = Permission({write: True, execute: False})
    self.donaturs[self.my_grandma_wallet] = 4000000000000000000

    donation_target : int128 = 80000000000000000000
```

When you deploy the smart contract to any blockchain, the `__init__` function of the Vyper file will be executed. Here, you can initialize state variables, the ones you defined outside functions.

Also, in the `initialization` function, commonly called the constructor, you also defined a local variable named `donation_target`. The difference is whether there is `self.` as a prefix or not for the variable.

Also, note that there is an `@external` annotation for the constructor. This means that this function can be called by the user. You can also make the function not be able to be called externally using the `@internal` annotation. Only functions in the same smart contract can call functions annotated with the `@internal` annotation.

You can compile the smart contract with the following command:

```
(.venv) $ vyper -f abi,bytecode StorageInit.vy
[{"stateMutability": "nonpayable", "type": "constructor", "inputs":
[],
 "outputs": []}]
0x3461010a5773de93510cfa39ab92bf927399f799dbe71997ee0b5f55600e6040527f
41726a756e6120536b79204b6f6b0000000000000000000000000000000000000606052
60408051602082018051600255508060015550506001600655600160075560026008055
60036009556005600a55600b546004811161010a57600181600c015560018101600b55
50600b546004811161010a57600181600c015560018101600b5550600b546004811161
010a57600281600c015560018101600b5550600160115555f601255673782dace9d9000
0060135f546020525f5260405f20556804563918244f40000060405261000d6100fc60
733961000d6073f3a1657679706572830003090000b005b5f80fd
```

This time, the ABI is not empty. Inside the ABI, you can find a way to interact with the smart contract. This particular function that interacts with the smart contract is called the constructor. It does not accept inputs and outputs in this case. Then, there is the `nonpayable` value in the `stateMutability` field. This means the constructor function does not accept ether payment, and it can change the value of state variables. We'll cover more in the next section.

Mutability

The annotations you've learned, `@external` and `@internal`, are for the visibility purpose. But there are other annotations with another purpose, such as **mutability**. Mutability means whether or not this function changes the state of the smart contract and receives the payment.

If a function doesn't change the state of the smart contract, you can use the `@pure` annotation or the `@view` annotation. The `@pure` annotation indicates that the function doesn't read the state variables and environment variables. We'll discuss the environment variable later. The `@view` annotation indicates that the function reads the state variables and environment variables but doesn't change the state variables.

If you want to write a function that changes the state variables, you can use the `@nonpayable` annotation and the `@payable` annotation. The difference between these two annotations is that functions with the `@payable` annotation can receive ethers or payment, while functions with the `@nonpayable` annotation cannot.

Yes, a program can receive and store money inside of the program. This is one of the unique properties of smart contracts. A smart contract can act like an escrow or a bank.

We will see an example of sending payments or ethers while executing a function that annotates with `@payable` in the next chapter.

Let's create a simple program to demonstrate these annotations. Create a file named `Annotations.vy` and add the following code to it:

```
# @version ^0.3.0

author: String[100]
donatur: String[100]

@external
def __init__():
    self.author = "Arjuna Sky Kok"

@external
@pure
def add(x: int128, y: int128) -> int128:
    return x + y
```

Here, we declared two state variables and then initialized one of them in the constructor function. Then, we created the `add` method that has the `@pure` annotation. Then, we added more functions at the bottom of the file:

```
@external
@view
def get_name_and_title() -> String[200]:
    return concat("Mr. ", self.author)

@external
@nonpayable
def change_name(new_name: String[100]):
    self.author = new_name

@external
@payable
def donate(donatur_name: String[100]):
    self.donatur = donatur_name
```

As you can see, the function with `@pure` doesn't read state variables at all. The `change_name` function changes the state variable author, but it cannot accept the payment. The `donate` function can accept the payment in ethers.

In the `get_name_and_title` function, you don't change the state variable, but you do read the state variable author. Inside the function, you used the `concat` method. This is one of the built-in functions you can use. There are a couple of built-in functions you can use for bitwise operations to manipulate data, like the one you used recently.

Let's compile the program by running the following command:

```
(.venv) $ vyper -f abi,bytecode Annotations.vy
[{"stateMutability": "nonpayable", "type": "constructor", "inputs":
[], "outputs": []}, {"stateMutability": "pure", "type": "function",
"name": "add", "inputs": [{"name": "x", "type": "int128"}, {"name":
"y", "type": "int128"}], "outputs": [{"name": "", "type": "int128"}]},
{"stateMutability": "view", "type": "function", "name": "get_name_and_
title", "inputs": [], "outputs": [{"name": "", "type": "string"}]},
{"stateMutability": "nonpayable", "type": "function", "name": "change_
name", "inputs": [{"name": "new_name", "type": "string"}], "outputs":
[]}, {"stateMutability": "payable", "type": "function", "name":
"donate", "inputs": [{"name": "donatur_name", "type": "string"}],
"outputs": []}]
0x3461029757600e6040527f41726a756e6120536b79204b6f6b000000....
```

Because you have a couple of functions, your ABI is becoming larger as well.

If you deploy this smart contract on Remix, you can send ethers when executing the `donate` function. Enter the amount of ethers you're going to send in the **Value** field first before executing the `donate` function, as shown in *Figure 3.6*.

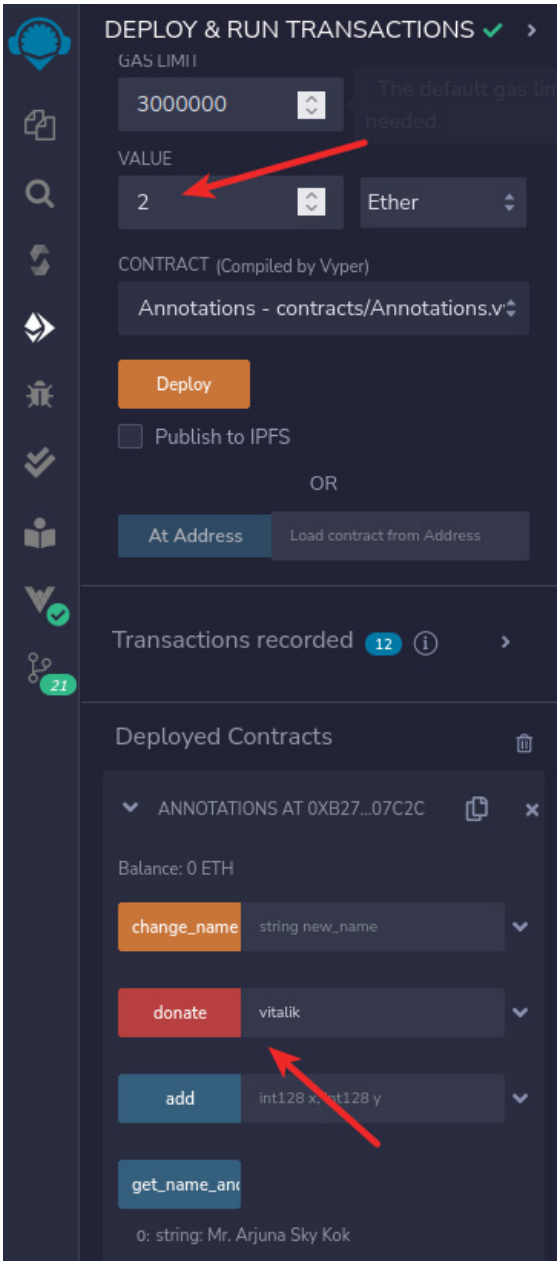


Figure 3.6: Executing the payable function

The payable function or the function where you can send ethers to the smart contract is marked in red. You can put the amount of ethers you want to send in the **Value** field.

Now, to make a smart contract really smart, we need control structures. That way, a smart contract has a way to decide what to do in certain situations.

Control structures

Just like the Python programming language, Vyper also has control structures, such as `if` and `for`, but with some restrictions compared to Python.

Let's create a Vyper file, named `ControlStructures.vy`, and add the following code to it:

```
# @version ^0.3.0

@external
@pure
def sum() -> int128:
    s: int128 = 0
    for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
        s += i
    return s

@external
@pure
def greet(time: String[10]) -> String[20]:
    if time == "morning":
        return "Good morning!"
    elif time == "evening":
        return "Good evening!"
    else:
        return "How are you?"
```

As you can see, the way you use `if` and `else` is similar to Python. However, there are some restrictions, such as not being able to modify the value of the array you iterate.

You can compile the program by using the following command:

```
(.venv) $ vyper -f abi,bytecode ControlStructures.vy
[{"stateMutability": "pure", "type": "function", "name": "sum",
"inputs": [], "outputs": [{"name": "", "type": "int128"}]},
{"stateMutability": "pure", "type": "function", "name": "greet",
"inputs": [{"name": "time", "type": "string"}], "outputs": [{"name":
"", "type": "string"}]}]
0x6102d161001161000396102d1610000f360033611...
```

Now that you understand control structures, it's time to understand the special variables that exist in a smart contract. These are crucial for operations in a smart contract, just like the `GET` request in a web application or environment variables in a console application.

Environment variables

There are environment variables that describe the block properties and the transaction. If you want to know which block you are in right now (remember that a smart contract lives in the blockchain), you can retrieve it from `block.number`. If you are keen to know the number of ethers that a user sends to a smart contract, you can use `msg.value`. If you are curious to know who sends money, you can query the information from `msg.sender`. If you want to store the time when a user sends ethers, you can use `block.timestamp`.

To demonstrate the environment variables, let's create a Vyper file, named `EnvVar.vy`, and add the following code to it:

```
# @version ^0.3.0

donatur: address
donation: uint256
time: uint256


@external
@payable
def donate():
    self.donatur = msg.sender
    self.donation = msg.value
    self.time = block.timestamp
```

When a user executes the `donate` function, they can send ethers or money to the smart contract. You can store the information in state variables. The `msg` environment variable refers to the transaction information. `block` refers to the block information.

You can compile the following program by using the following command:

```
(.venv) $ vyper -f abi,bytecode EnvVar.vy
[{"stateMutability": "payable", "type": "function", "name": "donate",
"inputs": [], "outputs": []}]
0x61003c61000f60003961003c6000f36003361161000c5761002b565b5f3560e01c
63ed88c68e81186100295733 5f5534600155426002550 05b505b5f5ffda165767970
657283000309000b
```

Although you now understand how to store data in a blockchain with state variables, sometimes, you will want to store an operation or an event. That's where event logging comes in.

Event logging

As well as storing information in state variables, you can also log information with event logging. This is useful for people who want to listen to some events. You may want to watch certain events on the blockchain, and if there is an event that you're interested in, you may want to do something, such as giving an announcement in a social media website that the donation goal has been reached.

To understand event logging, let's write a simple smart contract. Create a Vyper file, named `EventLogging.vy`, and add the following code to it:

```
# @version ^0.3.0

event Donation:
    donatur: indexed(address)
    amount: uint256


@external
@payable
def donate():
    log Donation(msg.sender, msg.value)
```

To log an event, you must declare an event that accepts what data you want to log. Here, you would want to log the `donatur` `indexed address` variable and the `uint256` `amount` variable. The indexed argument indicates that the variable can be searched by users. To log the event, you use `log` with the parameter, accompanied by the event name and its arguments. `msg.sender` will be logged to the `donatur` field of the `Donation` event, and the value of `msg.value` will be stored in the `amount` field of the `Donation` event.

So far, you've worked on a single smart contract, but it may need to interact with other smart contracts. To do that, you need an interface.

Interface

You have interacted with a smart contract as a user, but a smart contract can also interact with another smart contract using an interface.

Let's create a smart contract that interacts with the `Storage.vy` smart contract that you deployed previously, by creating `StorageClient.vy` and adding the following code to it:

```
# @version ^0.3.0

interface Storage:
    def retrieve() -> uint256: view

storage_contract: Storage
@external
def __init__(storage_address: address):
    self.storage_contract = Storage(storage_address)

@external
def call_retrieve() -> uint256:
    return self.storage_contract.retrieve()
```

You know that there is a `retrieve` method for `Storage.vy`. To interact with this method on `Storage.vy`, you create an interface, and then inside it, you define the method with the same name. You must also add the argument's signature and the return value. You can omit the definition of the `retrieve` method because the smart contract doesn't need to know. It only needs to know how to call the method and the return value.

Then, you create an object with the `Storage` interface type. In the `__init__` method, you initialize this object with the `address` variable. What address do you use? The address of the `Storage.vy` smart contract. Then, in the `call_retrieve` method, you demonstrate the calling of another smart contract's method.

The `__init__` method has an argument, so when you deploy the smart contract in Remix, you must fill in the address of `Storage.vy` in the field beside the **Deploy** button, as shown in *Figure 3.7*.

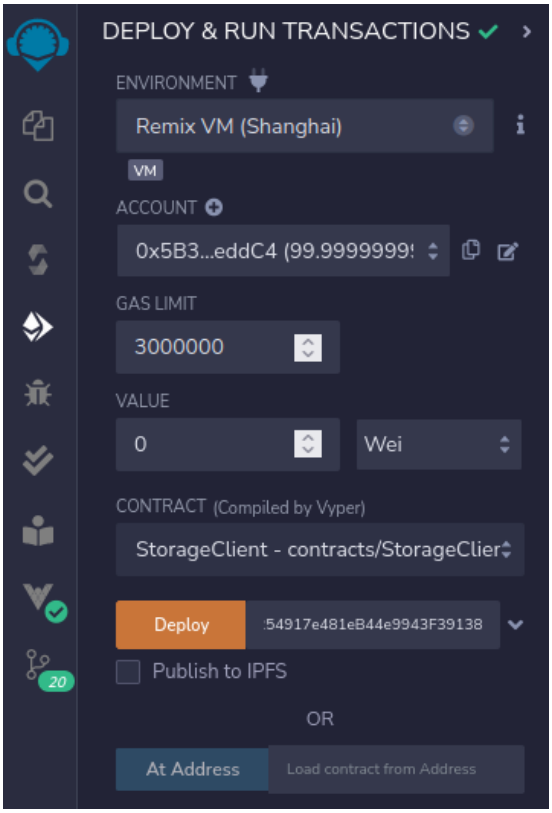


Figure 3.7: The address field beside the Aeploy button

When you deploy a smart contract that has `__init__` with arguments, you'll have options to provide the argument. The argument field is on the right side of the **Deploy** button.

Of course, in order to deploy `StorageClient.vy`, you must deploy `Storage.vy` first and get the address.

Summary

In this chapter, we started by diving into Vyper, the programming language that we're going to use to write smart contracts. First, we installed the Vyper compiler, and then we connected it to Remix to deploy a smart contract written in Vyper. Then, we took a look at all the features of Vyper, from data types, functions, control structures, and event logging to its interface. With this knowledge, you can start to write smart contracts with Vyper and deploy them to blockchain. You've become a smart contract engineer!

In the next chapter, we'll learn how to interact with smart contracts using Python.

Part 2: Web3 and Ape Framework

In this section, you'll get an overview of Web3 and Ape Framework that will equip you with the tools and frameworks necessary to interact with smart contracts and streamline the development of decentralized applications. You'll learn how to use the `Web3.py` library to connect to Ethereum nodes and invoke contract functions from Python. Additionally, this part introduces the powerful Ape development framework, which simplifies the process of building, testing, and deploying smart contracts. By the end of this section, you'll have the skills to build a practical decentralized application, solidifying your understanding of blockchain programming.

This section has the following chapters:

- *Chapter 4*, *Using Web3.py to Interact with Smart Contracts*
- *Chapter 5*, *Ape Framework*
- *Chapter 6*, *Building a Practical Decentralized Application*

4

Using Web3.py to Interact with Smart Contracts

In this chapter, we will learn how to use the **web3.py** library to deploy and interact with smart contracts. The `web3.py` library doesn't negate the necessity of writing smart contracts with Vyper or Solidity. In the previous chapter, you deployed smart contracts and interacted with smart contracts with Remix IDE. What the `web3.py` library does is similar to Remix IDE, so much so that you can completely replace Remix IDE with the `web3.py` library.

The following topics will be covered in this chapter:

- Installing `web3.py`
- Deploying a smart contract
- Interacting with smart contracts

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_4`.

Installing web3.py

The **web3.py library** is a Python library. So, you could install it the way you install any Python library. The recommended way is to use a Python virtual environment.

First, create a Python virtual environment:

```
$ python3.10 -m venv .venv
$ source .venv/bin/activate
```

Then, install the library like so:

```
(.venv) $ pip install web3
```

Now that you've installed the `web3.py` library, it's time to test it with blockchains. We're going to test it with development blockchains. In the previous chapter, you used the development blockchain embedded in Remix IDE. Although it lives in the browser, there are development blockchains that run outside the browser.

Ganache

Ganache is touted as the personal Ethereum blockchain. Using this application, you can easily experiment with the blockchain. One of the unique features of this application is that it has a graphical interface. The other blockchains usually only have a command-line interface.

The recommended replacement for Ganache is Hardhat, `https://hardhat.org`. But for learning purposes, it's fine to use Ganache.

Ganache is available for Windows, Linux, and Mac. However, we're only going to explain the installation process in Linux. Go to `https://trufflesuite.com/ganache/`. You'll be presented with the official Ganache website, as shown in *Figure 4.1*:

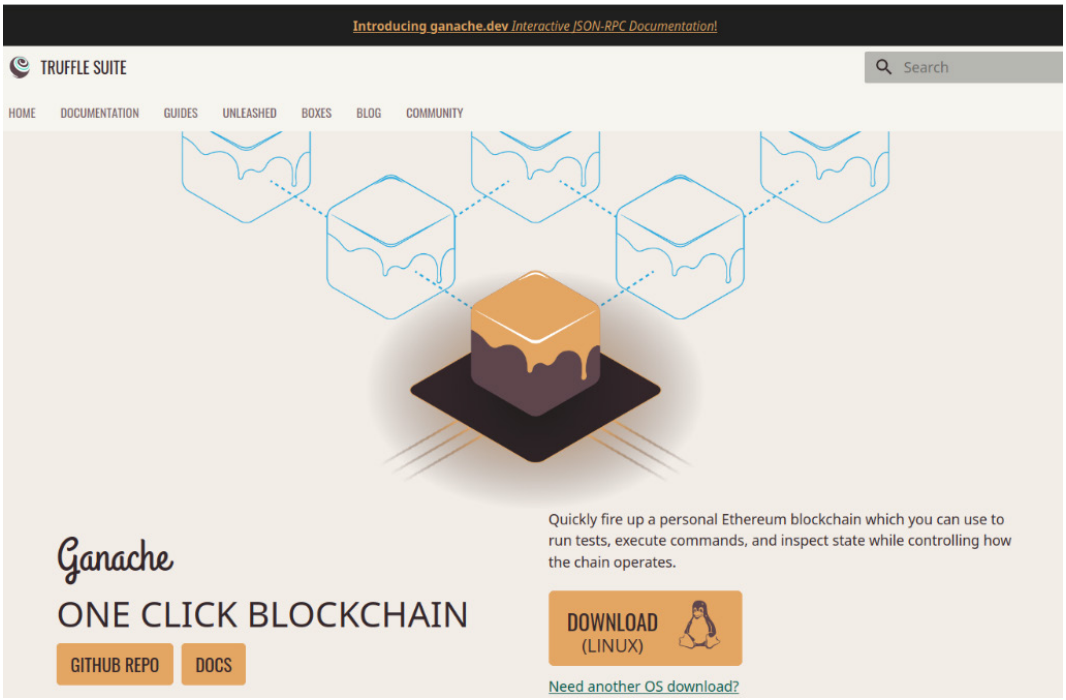


Figure 4.1: The Ganache website

Then, click the **DOWNLOAD (LINUX)** button. You'll receive a file named `ganache-2.7.1-linux-x86_64.AppImage`. Your file version might be different if there is a new version of Ganache available at the time you're reading this.

Make the file executable by running the following command:

```
$ chmod +x ganache-2.7.1-linux-x86_64.AppImage
```

Then, you can run Ganache with the following command:

```
$ ./ganache-2.7.1-linux-x86_64.AppImage
```

You will see the introduction screen shown in *Figure 4.2*:

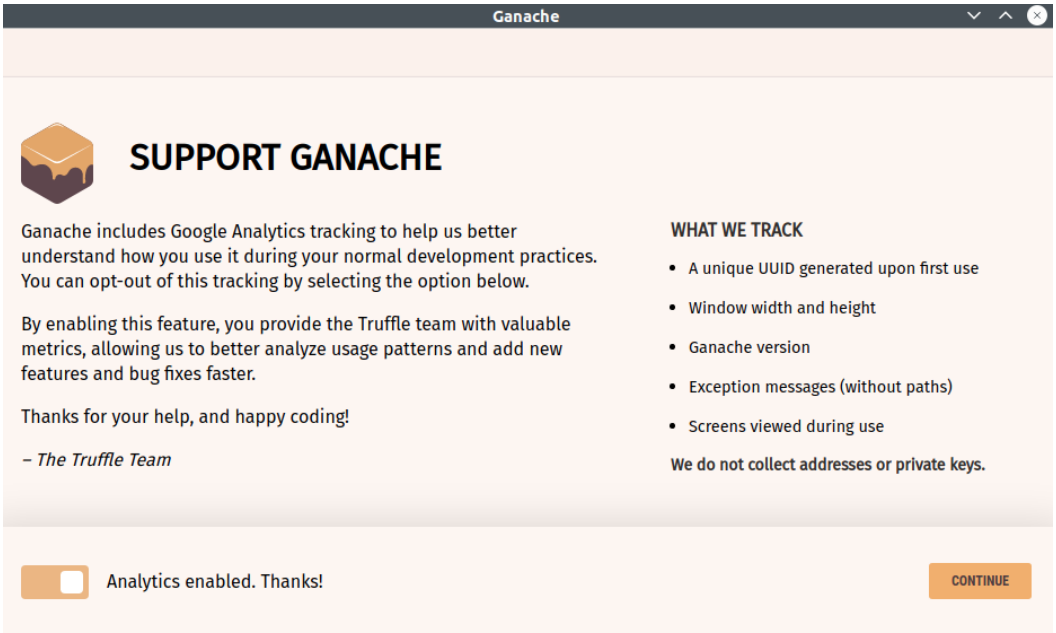


Figure 4.2: Ganache introduction screen

Click the **CONTINUE** button. You will be taken to the following screen:

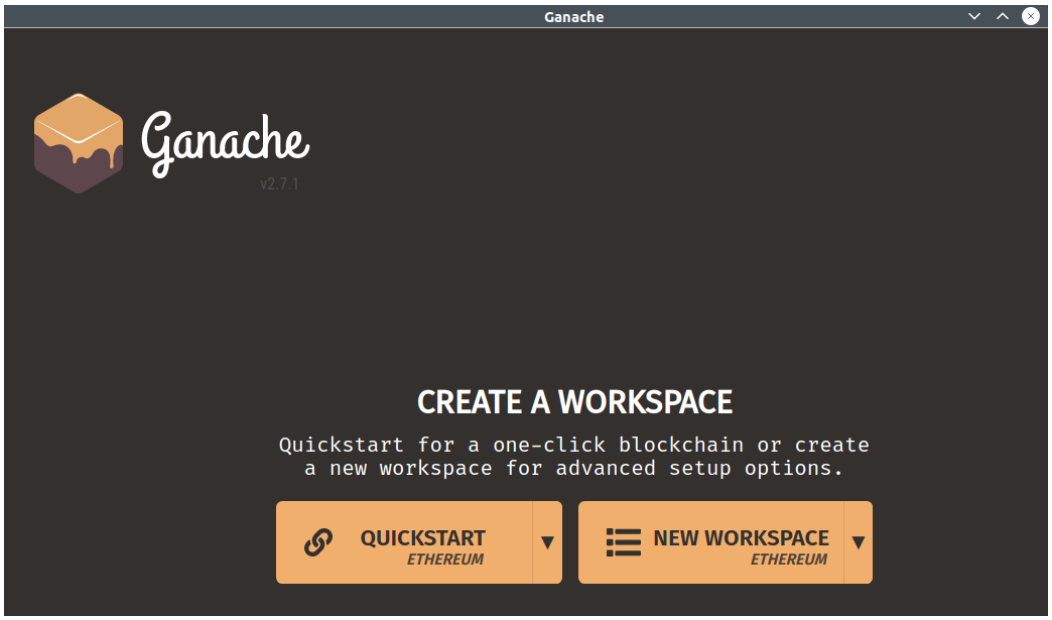


Figure 4.3: The Ganache quickstart screen

Here, you can choose to create a workspace. In Ganache, a workspace is a space for the blockchain. So, if you want to create a blockchain for development purposes and another blockchain for testing purposes, you could create two separate workspaces.

Click the **QUICKSTART** button. You will be greeted with the screen shown in *Figure 4.4*:

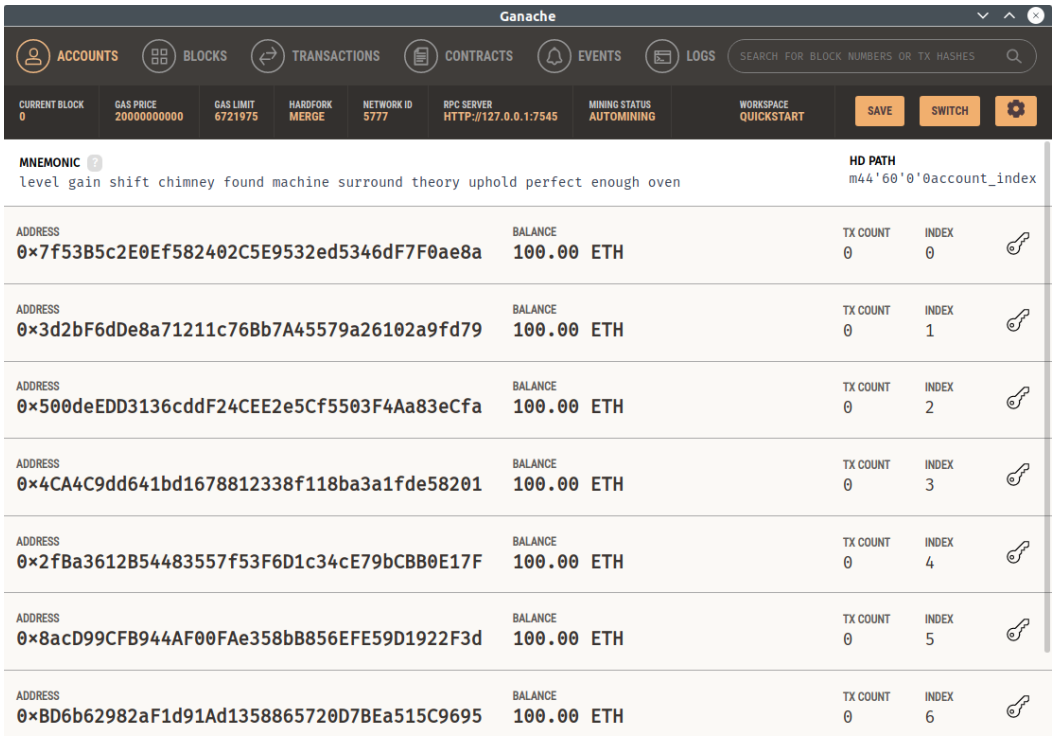


Figure 4.4: Ganache accounts screen

On this screen, you'll see a couple of accounts. Each account is bestowed with 100 ETH. The workspace is named `QUICKSTART`. The current block is 0, meaning this is the very beginning of this blockchain history. The RPC server is located at `http://127.0.0.1/7545`. This is an important piece of information. When you want to connect to the blockchain, you need to find where to connect. The RPC server is one way to connect to the blockchain.

Look at the third account from the top. The address is `0x500deEDD3136cddF24CEE2e5Cf5503F4Aa83eCfa`. We're going to use this address in a script. However, yours might be different. The generated addresses in Ganache are random. So, you need to adjust it accordingly.

Let's write a Python script to connect to this blockchain. Create a file named `hello_web3.py` and add the following code to it:

```
from web3 import Web3

w3 = Web3(Web3.HTTPProvider('http://127.0.0.1:7545'))

print(f"Python script is connected? {w3.is_connected()}")

# The balance of the third account. Change it according to generated
addresses on your Ganache.
account_balance = w3.eth.get_
balance('0x500deEDD3136cddF24CEE2e5Cf5503F4Aa83eCfa')

print(f"The balance of the 3rd account is {account_balance}")
```

In the preceding script, you created a connection object to the blockchain named `w3`. Notice that you passed the RPC provider URL on the Ganache screen, `http://127.0.0.1:7545`, to the `Web3.HTTPProvider` method.

Once you've created the connection object, you can test whether it's connected or not with the `w3.is_connected` method.

Once you've confirmed it's connected, you can query the information on the blockchain to find out how much ETH an account has on this blockchain with the `w3.eth.get_balance` method. This method accepts the address that you took from the third account on the Ganache screen.

Run the Python script:

```
(.venv) $ python hello_web3.py
Python script is connected? True
The balance of the 3rd account is 100000000000000000000
```

When you look at the Ganache screen, you'll see 100 ETH as the balance of the account. The `get_balance` method returns the balance in **wei**.

> **Note**
>
> 1 ETH is 1,000,000,000,000,000,000 wei (18 zeros). So, 100 ETH is 100,000,000,000,000,000,000. Think of ETH as a dollar and wei as a cent.

As you know, to do something with any account in the blockchain, you need to know it's private key. It's like the password to your account. In Ganache, to find out the private key of an account, you can click the key icon of the account, as shown in *Figure 4.6*:

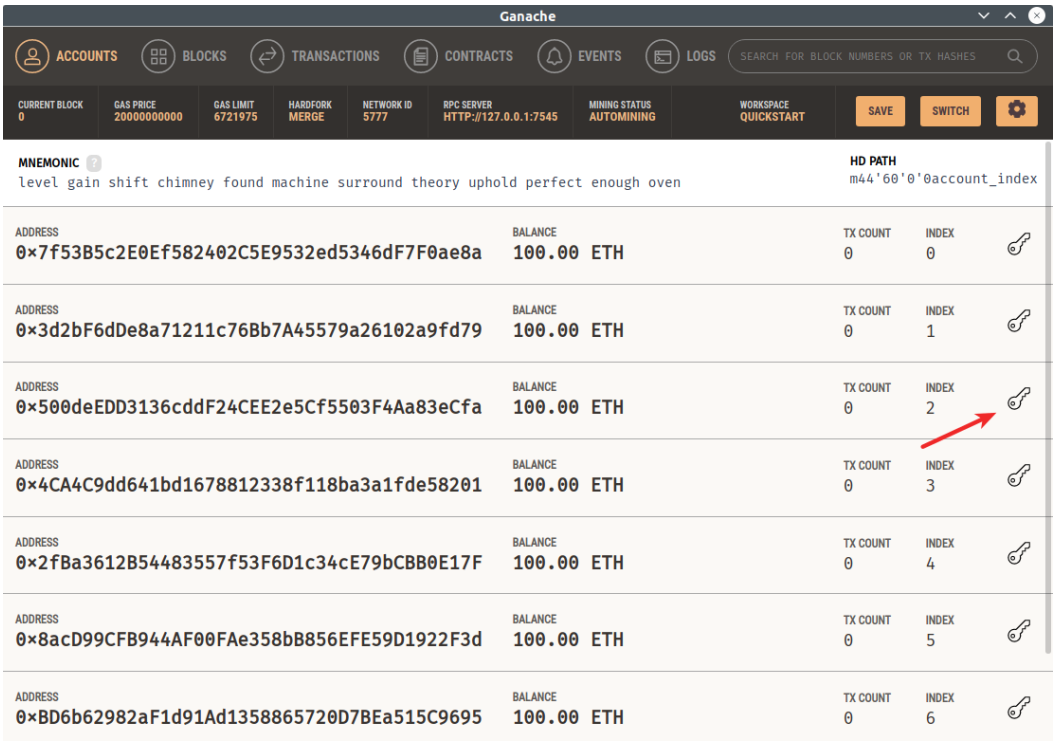


Figure 4.5: The key icon in Ganache

Click the key icon; you'll see the private key of that account, as shown in *Figure 4.6*:

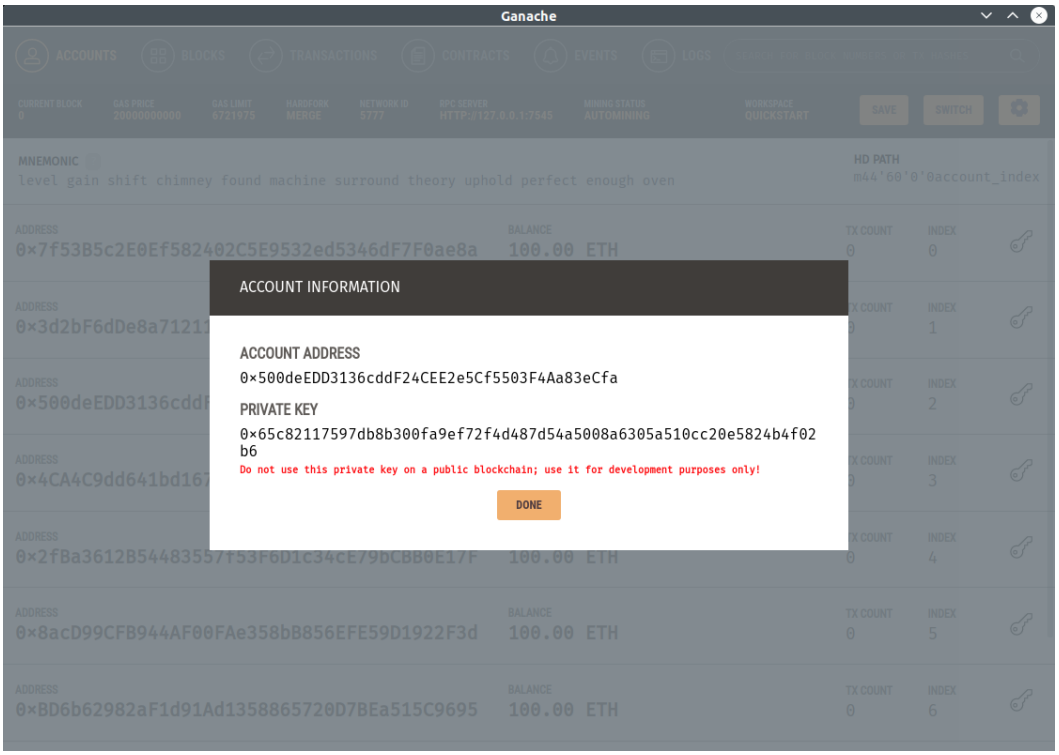


Figure 4.6: The private key window

The private key for the third account, `0x500deEDD3136cddF24CEE2e5Cf5503F4Aa83eCfa`, is `0x65c82117597db8b300fa9ef72f4d487d54a5008a6305a510cc20e5824b4f02b6`.

Since your third account is different, the private key will be different too. The recipient's address will also be different because the address is taken from the fourth account on Ganache. Remember that the generated addresses are random. So, adjust the following script accordingly.

Let's create a script that will send some ETHs from this account to another account. Create a file named `send_eth.py` and add the following content to it:

```
from web3 import Web3

w3 = Web3(Web3.HTTPProvider('http://127.0.0.1:7545'))

sender_private_key =
"0x65c82117597db8b300fa9ef72f4d487d54a5008a6305a510cc20e5824b4f02b6"
```

```
sender_address = "0x500deEDD3136cddF24CEE2e5Cf5503F4Aa83eCfa"

recipient_address = "0x4CA4C9dd641bd1678812338f118ba3a1fde58201"

amount = Web3.to_wei(5, 'ether')
```

In the preceding code, you began by declaring the address of the recipient and the sender. After that, you added the private key of the sender. Then, you specified the amount of ETH you wanted to send from the sender to the recipient. When you create Python scripts dealing with blockchain, you always use the lowest measurement unit, which is wei. However, often, ETH is a more familiar measurement unit. So, you could use the `Web3.to_wei` method to convert from one measurement unit into another.

Add the following code:

```
transaction = {
    'from': sender_address,
    'to': recipient_address,
    'value': amount,
    'gas': 21000,
    'gasPrice': Web3.to_wei('50', 'gwei'),
    'nonce': w3.eth.get_transaction_count(sender_address),
}

signed_transaction = w3.eth.account.sign_transaction(transaction,
sender_private_key)

transaction_hash = w3.eth.send_raw_transaction(signed_transaction.
rawTransaction)

print(transaction_hash)
```

The sending ETH transaction can be defined with the help of a dictionary comprising the sender, the destination, and the amount. In finance, sometimes, you have to pay a fee to payment providers when making a payment. It's the same as in blockchain. The fee is represented by `gas` and `gasPrice`.

Here, `gas` and `gasPrice` are needed to pay the fee. It is important to be aware of the fact that transaction on the blockchain is not free. You pay some fees to the validators. The amount of `gas` is linked to the complexity of the transaction. For sending ETH, 21,000 is enough. But other complex transactions require more gas. So, `gasPrice` depends on how jammed the blockchain network is. If many people use the blockchain at a particular moment, `gasPrice` can be more expensive. If you use a lower `gasPrice` value, your transaction could be rejected by validators.

The nonce's purpose is to protect the transaction being repeated. To create a transaction, you must get the nonce from how many transactions you've created on the blockchain. If you've created three transactions and you want to create a new transaction, you should use 3 as the value for nonce.

Then, you signed the transaction with `w3.eth.account.sign_transaction`, which accepts the transaction as the first argument and the private key as the second argument. If you want to send money from A to B, you need to know the private key of A, but you don't need to know the private key of B.

Finally, after creating the signed transaction, you sent it to the blockchain with the `w3.eth.send_raw_transaction` method.

Let's execute the script and send 5 ETH from one account to another:

```
$ python send_eth.py
b'\xb3\x02\xad\xb3\x8fj\xe9\xa7\x01\x1c\x1a\xe8\\\xe8\xd9,t\x11)$J\\\
x19\xf7\xa0\xd3!\xfds\xac/f'
```

The output is the transaction hash, which is the transaction ID. It's similar to a confirmation of your transaction being sent to the blockchain. You can use this transaction hash to locate the transaction on the blockchain and see whether it has been confirmed or not on the blockchain.

Let's look at the Ganache screen:

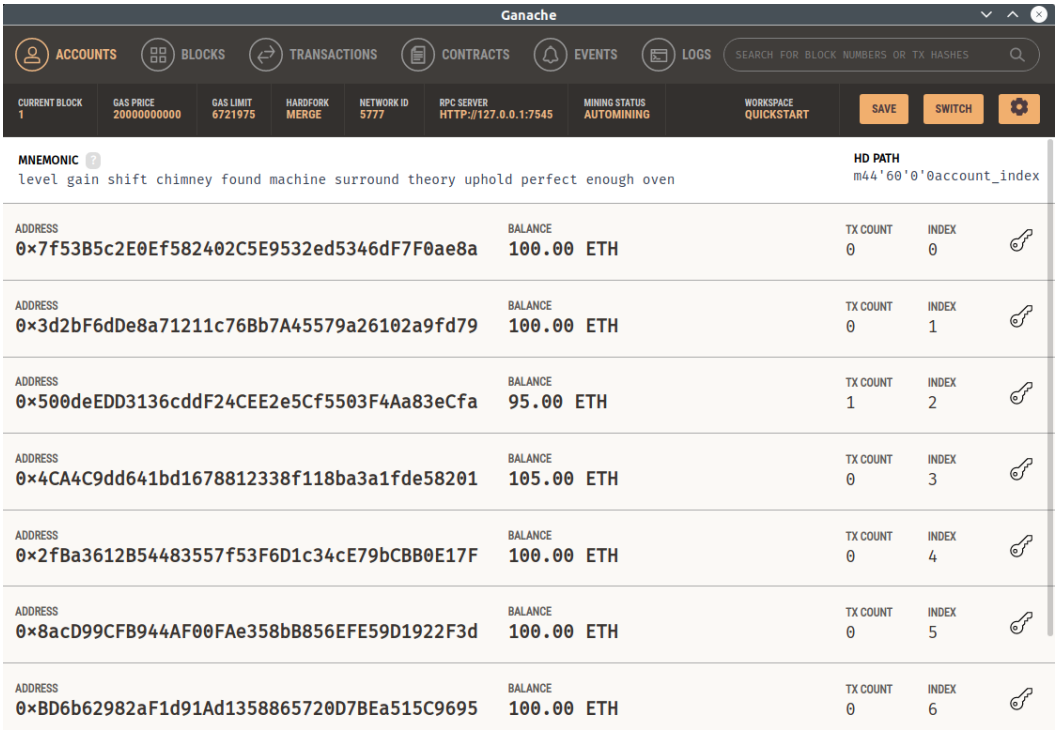


Figure 4.7: The changing balance after the transaction

Notice that the third account has 95 ETH and the fourth account has 105 ETH. This means that you've successfully transferred ETH with a Python script!

Geth

Another blockchain you could install is **Go-Ethereum** (**Geth**). It's available for Windows, Mac, and Linux. Note that we'll only show you how to install Geth on Linux in this chapter.

Run the following command:

```
$ sudo add-apt-repository -y ppa:ethereum/ethereum
$ sudo apt update
$ sudo apt-get install ethereum
```

To run the development blockchain with Geth, run the following command:

```
$ geth --dev --http --http.api eth,web3,net
INFO [07-09|00:40:44.479] Starting Geth in ephemeral dev mode...
WARN [07-09|00:40:44.479] You are running Geth in --dev mode. Please
note the following:


  1. This mode is only intended for fast, iterative development
without assumptions on
     security or persistence.
  2. The database is created in memory unless specified otherwise.
Therefore, shutting down
     your computer or losing power will wipe your entire block data
and chain state for
     your dev environment.
  3. A random, pre-allocated developer account will be available and
unlocked as
     eth.accounts[0], which can be used for testing. The random dev
account is temporary,
     stored on a ramdisk, and will be lost if your machine is
restarted.
  4. Mining is enabled by default. However, the client will only seal
blocks if transactions
     are pending in the mempool. The miner's minimum accepted gas
price is 1.
  5. Networking is disabled; there is no listen-address, the maximum
number of peers is set
     to 0, and discovery is disabled.
...
```

This blockchain runs in memory, and all transactions on this blockchain will cease to exist if the process ends.

If you scroll through the command's output, you'll find important information:

```
INFO [07-09|00:40:44.657] Chain ID:  1337 (unknown)
INFO [07-09|00:40:44.657] Consensus: Clique (proof-of-authority)
INFO [07-09|00:40:44.657]
INFO [07-09|00:40:44.657] Pre-Merge hard forks (block based):
INFO [07-09|00:40:44.657]  -
Homestead:                    #0        (https://github.com/ethereum/
execution-specs/blob/master/network-upgrades/mainnet-upgrades/
homestead.md)
INFO [07-09|00:40:44.657]  - Tangerine Whistle (EIP 150):
#0        (https://github.com/ethereum/execution-specs/blob/master/
network-upgrades/mainnet-upgrades/tangerine-whistle.md)
INFO [07-09|00:40:44.657]  - Spurious Dragon/1 (EIP 155):
#0        (https://github.com/ethereum/execution-specs/blob/master/
network-upgrades/mainnet-upgrades/spurious-dragon.md)
INFO [07-09|00:40:44.657]  - Spurious Dragon/2 (EIP 158):
#0        (https://github.com/ethereum/execution-specs/blob/master/
network-upgrades/mainnet-upgrades/spurious-dragon.md)
INFO [07-09|00:40:44.657]  -
Byzantium:                    #0        (https://github.com/ethereum/
execution-specs/blob/master/network-upgrades/mainnet-upgrades/
byzantium.md)
INFO [07-09|00:40:44.657]  -
Constantinople:               #0        (https://github.com/ethereum/
execution-specs/blob/master/network-upgrades/mainnet-upgrades/
constantinople.md)
INFO [07-09|00:40:44.657]  -
Petersburg:                   #0        (https://github.com/ethereum/
execution-specs/blob/master/network-upgrades/mainnet-upgrades/
petersburg.md)
INFO [07-09|00:40:44.657]  -
Istanbul:                     #0        (https://github.com/ethereum/
execution-specs/blob/master/network-upgrades/mainnet-upgrades/
istanbul.md)
INFO [07-09|00:40:44.657]  - Muir
Glacier:                 #0        (https://github.com/ethereum/
execution-specs/blob/master/network-upgrades/mainnet-upgrades/muir-
glacier.md)
INFO [07-09|00:40:44.657]  -
Berlin:                       #0        (https://github.com/ethereum/
execution-specs/blob/master/network-upgrades/mainnet-upgrades/berlin.
md)
INFO [07-09|00:40:44.657]  -
London:                       #0        (https://github.com/ethereum/
execution-specs/blob/master/network-upgrades/mainnet-upgrades/london.
md)
```

Notice that `chainId` is `1337`. You will need this later in this chapter.

Going down, you'll find out how to connect to this blockchain:

```
INFO [07-09|00:40:44.664] IPC endpoint opened
        url=/tmp/geth.ipc
INFO [07-09|00:40:44.664] Generated ephemeral JWT
secret           secret=0xbab5ce00950f406b435d314eb98e975599
8fb545e6783ea4604ed9177fc0e2a4
INFO [07-09|00:40:44.665] HTTP server started
       endpoint=127.0.0.1:8545 auth=false prefix= cors=
vhosts=localhost
INFO [07-09|00:40:44.665] WebSocket enabled
       url=ws://127.0.0.1:8551
INFO [07-09|00:40:44.665] HTTP server started
       endpoint=127.0.0.1:8551 auth=true  prefix= cors=local
host vhosts=localhost
```

You could connect to the blockchain via HTTP by using `http://127.0.0.1:8545`. But there are other options as well, such as using **inter-process communication** (**IPC**) and WebSockets. To connect with IPC, you must get the location of the IPC endpoint file. As seen in the output message, the location is `/tmp/geth.ipc`. However, on other platforms, the location will be different.

Geth is a server as well as a client application. You could connect to this blockchain with Geth. Let's connect to the blockchain using IPC, a protocol you can use to communicate between processes on an operating system. Run the following command:

```
$ geth attach /tmp/geth.ipc
Welcome to the Geth JavaScript console!

instance: Geth/v1.11.3-stable-5ed08c47/linux-amd64/go1.20.2
coinbase: 0xd30561464ee30ff007e8e1c49aa950448db485bd
at block: 0 (Thu Jan 01 1970 07:00:00 GMT+0700 (WIB))
datadir:
modules: admin:1.0 clique:1.0 debug:1.0 engine:1.0 eth:1.0 miner:1.0
net:1.0 rpc:1.0 txpool:1.0 web3:1.0

To exit, press ctrl-d or type exit
>
```

Notice that you give the location of the IPC endpoint file to `geth`. You're in a Geth prompt. You can run some commands here, such as the following, to see what development accounts are available to you:

```
> eth.accounts
["0xd30561464ee30ff007e8e1c49aa950448db485bd"]
```

Unlike Ganache, the Geth development blockchain only gives you one testing account. The generated address is random, so you will see a different address.

Let's run this command to check how much ETH this testing account has:

```
> web3.fromWei(eth.getBalance(eth.accounts[0]))
1.157920892373161954235709850086879078532699846656405640394575840079131
29639927e+59
```

As you can see, this is a very large amount. It's not 1.15 ETH, but 1.15e+59 (it has 59 zeros).

To create another testing account, you can use `clef`. It comes from the same package as `geth`. Create a directory, like so:

```
$ mkdir my_keystore
```

Now, run the following command:

```
$ clef newaccount --keystore my_keystore

WARNING!

Clef is an account management tool. It may, like any software, contain
bugs.

Please take care to
- backup your keystore files,
- verify that the keystore(s) can be opened with your password.

Clef is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY;
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR
PURPOSE. See the GNU General Public License for more details.

Enter 'ok' to proceed:
>
```

Run the `clef` command. As we can see, it will keep the account data in `my_keystore`. Type `ok` and press *Enter*:

```
> ok

## New account password

Please enter a password for the new account to be created (attempt 0
of 3)
>
```

Then, type a password to protect your private key. Your password must not be too simple. After typing the password, you'll get the following message:

```
INFO [07-09|01:06:13.287] Your new key was generated
address=0xA7603e35744EA8d88F5F66f0dE7bcE24f2Bf8F2f
WARN [07-09|01:06:13.287] Please backup your key file!
path=/home/arjuna/Code/books/Hands-On-Blockchain-for-Python-Developers
--2nd-Edition/chapter_4/my_keystore/UTC--2023-07-08T18-06-
12.161925018Z
--a7603e35744ea8d88f5f66f0de7bce24f2bf8f2f
WARN [07-09|01:06:13.287] Please remember your password!
Generated account 0xA7603e35744EA8d88F5F66f0dE7bcE24f2Bf8F2f
```

Your generated account will be different from `0xA7603e35744EA8d88F5F66f0dE7bcE24f2Bf8F2f`. The file that holds the private key is in `my_keystore/ UTC--2023-07-08T18-06-12.161925018Z--a7603e35744ea8d88f5f66f0de7bce24f2bf8f2f`. Your result file will be different as well. So, change the address when running the next few commands.

Now, let's go back to the `geth` prompt. You can send money to this new account like so:

```
> eth.sendTransaction({from: eth.accounts[0], to:
"0xA7603e35744EA8d88F
5F66f0dE7bcE24f2Bf8F2f", value: web3.toWei(10, "ether")})
"0x3e309961ba01ecf2a648f6a6eca226e001bfb537f92068657a4124ec4bfb27a1"
```

Now, check the balance of the new account:

```
> web3.fromWei(eth.
getBalance("0xA7603e35744EA8d88F5F66f0dE7bcE24f2Bf8F2f"))
10
```

Make sure you change `0xA7603e35744EA8d88F5F66f0dE7bcE24f2Bf8F2f` to your generated account.

Let's create a Python script that can send 2 ETH from this new account. Create a file named `send_eth_on_geth.py` and add the following code to it:

```
from web3 import Web3

w3 = Web3(Web3.IPCProvider('/tmp/geth.ipc'))

with open("my_keystore/UTC--2023-07-08T18-06-12.161925018Z--a7603e3574
4ea8d88f5f66f0de7bce24f2bf8f2f") as keyfile:
    encrypted_key = keyfile.read()
    password = "mypassword123"
    sender_private_key = w3.eth.account.decrypt(encrypted_key,
password)
```

```
sender_address = "0xA7603e35744EA8d88F5F66f0dE7bcE24f2Bf8F2f"

recipient_address = Web3.to_checksum_
address("0xd30561464ee30ff007e8e1c49aa950448db485bd")

amount = Web3.to_wei(2, 'ether')
```

Remember to change the location of the IPC provider, `/tmp/geth.ipc`, to your geth.ipc's location. Also, make sure to change the keyfile's location, the password, and the sender address. You don't have to change the recipient's address.

Unlike the previous script, you connected to the blockchain using the `Web3.IPCProvider` method. Then, instead of providing the private key in the code, you unlocked it from the encrypted file you generated using `clef`. To unlock it, you used the password you provided. In my case, I used `mypassword123`.

Then, you provided the sender address (your generated account), the recipient address, and the amount of ETH you want to send. Notice that the recipient address is put inside the `Web3.to_checksum_address` method. This ensures that the address is valid. It's useful for you so that you don't send ETH to non-existent addresses. The `Web3.to_checksum_address` method will throw an exception if you use an invalid address, such as "my grandma's address."

The script isn't complete yet. Add the following code to the bottom of the file:

```
transaction = {
    'chainId': 1337,
    'from': sender_address,
    'to': recipient_address,
    'value': amount,
    'gas': 21000,
    'gasPrice': Web3.to_wei('50', 'gwei'),
    'nonce': w3.eth.get_transaction_count(sender_address),
}

signed_transaction = w3.eth.account.sign_transaction(transaction,
sender_private_key)

transaction_hash = w3.eth.send_raw_transaction(signed_transaction.
rawTransaction)

print(transaction_hash)
```

Notice that you can add `chainId` to the `transaction` object. You can omit `chainId` when using Ganache but not with Geth. There are many blockchains that you will encounter; to differentiate them, you can use `chainId`.

Run the following script:

```
(.venv) $ python send_eth_on_geth.py
b'\xdf\x8a\xda7W\xbdg\xc9M\n\xfdE\xc3D\xa6\xd4\x97\x15\xcf&4\xfa\xb1\
xcf\xb1\x17\xe7\x86\xac\xdc\x00\xe3'
```

Now, go back to the Geth prompt and check your generated account again:

```
> web3.fromWei(eth.
getBalance("0xA7603e35744EA8d88F5F66f0dE7bcE24f2Bf8F2f"))
7.99895
```

You lost 2 ETH because you sent it to another account. The end balance of your account is not exactly 8 ETH. It's 7.99895 ETH because you must pay the gas fee. The gas and its price, when combined, will become the transaction fees you pay.

However, `web3.py` can do more than just send ETH. It can also interact with and deploy smart contracts.

Deploying a smart contract

To deploy a smart contract, you need a smart contract first. So, let's create a smart contract. Create a file named `SimpleContract.vy` and add the following code to it:

```
# @version ^0.3.0

event Donation:
    donatur: indexed(address)
    amount: uint256

num: uint256

@external
def store(num: uint256):
    self.num = num

@external
def retrieve() -> uint256:
    return self.num

@external
@payable
def donate():
    log Donation(msg.sender, msg.value)
```

This smart contract has a method named `retrieve` to get the value of the `num` state variable. It also has a method named `store` to change the value of the `num` state variable. Lastly, it has a method named `donate` to accept ETH.

Now, you need to compile the smart contract into the bytecode and the ABI. Let's save them in files. To do this, first, you need to install Vyper:

```
(.venv) $ pip install vyper==0.3.10
```

You can save the bytecode from the compilation process:

```
(.venv) $ vyper SimpleContract.vy -f bytecode > bytecode.txt
```

You will also need the ABI. Save the ABI from the complication process:

```
(.venv) $ vyper SimpleContract.vy -f abi > abi.json
```

To ensure that you've saved the bytecode and ABI correctly, you can check the content of the files:

```
(.venv) $ cat bytecode.txt
0x6100ab61000f6000396100ab6000f36003361161000c57610093565b60003560e01c
63ed88c68e81186100525760043610610099573 7f5d8bc849764969eb1bcc6d0a2f55
999d0167c1ccec240a4f39cf664ca9c4148e3460405260206040a2005b346100995763
6057361d81186100725760243610610099576004356000550 05b632e64cec181186100
91576004361061009957600054604052602060 40f35b505b60006000fd5b600080fda1
65767970657283000307000b
(.venv) cat abi.json
[{"name": "Donation", "inputs": [{"name": "donatur", "type":
"address", "indexed": true}, {"name": "amount", "type": "uint256",
"indexed": false}], "anonymous": false, "type": "event"},
{"stateMutability": "nonpayable", "type": "function", "name": "store",
 "inputs": [{"name": "num", "type": "uint256"}], "outputs": []},
{"stateMutability": "nonpayable", "type": "function", "name":
"retrieve", "inputs": [], "outputs": [{"name": "", "type":
"uint256"}]}, {"stateMutability": "payable", "type": "function",
"name": "donate", "inputs": [], "outputs": []}]
```

Now, the stage is set for us to create a Python script to deploy this smart contract. Create a file named `deploy.py` and add the following code to it:

```
from web3 import Web3

w3 = Web3(Web3.IPCProvider('/tmp/geth.ipc'))

with open("bytecode.txt") as f:
    bytecode = f.read().strip()
```

```
with open("abi.json") as f:
    abi = f.read().strip()

with open("my_keystore/UTC--2023-07-08T18-06-12.161925018Z--a7603e3574
4ea8d88f5f66f0de7bce24f2bf8f2f") as keyfile:
    encrypted_key = keyfile.read()
    password = "mypassword123"
    deployer_private_key = w3.eth.account.decrypt(encrypted_key,
password)

deployer_address = "0xA7603e35744EA8d88F5F66f0dE7bcE24f2Bf8F2f"
```

With the preceding code, you read the bytecode and the ABI back from the files. Then, you declared the private key and the address of the deployer. In this example, you decided to deploy a smart contract to the Geth blockchain.

You also need to change the location of the IPC file, the keyfile's location, the password, and the deployer address since yours will be different.

Add the following code to the bottom of the file:

```
contract = w3.eth.contract(abi=abi, bytecode=bytecode)

transaction = contract.constructor().build_transaction({
    'from': deployer_address,
    'nonce': w3.eth.get_transaction_count(deployer_address),
    'gas': 200000,
    'gasPrice': Web3.to_wei('50', 'gwei')
})

signed_transaction = w3.eth.account.sign_transaction(transaction,
deployer_private_key)

transaction_hash = w3.eth.send_raw_transaction(signed_transaction.
rawTransaction)

print(transaction_hash)
```

To create a smart contract object with `web3`, you can use the `w3.eth.contract` method. You've already provided the ABI and the bytecode. You don't need the source code of the smart contract itself.

Just like creating a sending ETH transaction, you needed to create a transaction to deploy a smart contract. In this case, you used the `contract.constructor().build_transaction` method to create a transaction. The `constructor` method refers to the `__init__` method in the Vyper file (if it exists). In this case, there is no `__init__` method, so it uses the default constructor, which doesn't accept any argument. Because of that, you executed the method like this: `constructor()`. However, if there is a constructor method in the Vyper file that accepts arguments, you need to provide it.

This `transaction` object is similar to the `transaction` object in the sending ETH transaction. You provided `from`, `nonce`, `gas`, and `gasPrice`. Here, `to` is missing because you don't need it.

Notice that you provided more gas compared to the previous transactions. Hence, we can say that deploying a smart contract is more complex than sending money.

The rest of the code is similar to the previous scripts. All you need to do is sign and send the transaction.

Run the script:

```
(.venv) $ python deploy.py
b"\xac_6\x1b\xec_\x8cu\xac\xd3\x84\xb1r\r\xe9\xc8W\xb9\xde\x1a'
\xc6\x9b1G\x8f\x8f\xa3a\xfe!\xec"
```

Your smart contract has already been deployed on the Geth blockchain. You can peek at the address of the deployed smart contract on Geth log messages in the terminal where you run the Geth server:

```
INFO [07-09|21:25:52.199] Submitted contract creation
hash=0xac5f361bec5f8c75acd384b1720de9c857b9de1a27c69b31478f8fa361fe
21ec from=0xA7603e35744EA8d88F5F66f0dE7bcE24f2Bf8F2f nonce=6 contract
=0x8bb44f25E5b25ac14c8A9f5BFAcdFd1a700bA18B value=0
INFO [07-09|21:25:52.199] Commit new sealing work
number=8 sealhash=280a79..145b72 uncles=0 txs=1 gas=89942 fees=0.0044
65832819 elapsed="277.461µs"
INFO [07-09|21:25:52.199] Successfully sealed new block
number=8 sealhash=280a79..145b72 hash=7e01d9..b4bd0e
elapsed="324.945µs"
INFO [07-09|21:25:52.199] "🔗 block reached canonical chain"
number=1 hash=0f62cb..a9d8b9
INFO [07-09|21:25:52.199] "🔨 mined potential block"
number=8 hash=7e01d9..b4bd0e
```

The address of the smart contract is `0x8bb44f25E5b25ac14c8A9f5BFAcdFd1a700bA18B`.

Before interacting with the deployed smart contract, how do you know how much gas is needed? You can estimate this with `web3.py`. Create a file named `estimate_gas.py` and add the following code to it:

```
from web3 import Web3

with open("bytecode.txt") as f:
    bytecode = f.read().strip()
```

```
with open("abi.json") as f:
    abi = f.read().strip()
deployer_address = "0x500deEDD3136cddF24CEE2e5Cf5503F4Aa83eCfa"

w3 = Web3(Web3.HTTPProvider('http://127.0.0.1:7545'))

contract = w3.eth.contract(abi=abi, bytecode=bytecode)

total_gas = contract.constructor().estimate_gas({
    'from': deployer_address,
    'nonce': w3.eth.get_transaction_count(deployer_address),
    'gas': 200000,
    'gasPrice': Web3.to_wei('50', 'gwei')
})

print(total_gas)
```

Make sure you change the deployer address in the script to your deployer address.

Run the script to get the estimated gas needed to deploy a simple smart contract:

```
(.venv) $ python estimate_gas.py
89942
```

As you can see, you need around 89k gas. You could add more gas just to be safe. Unused gas, however, will be sent back to the deployer account. If you don't have enough gas, your transaction will fail.

Now that you have deployed the smart contract, let's interact with it.

Interacting with smart contracts

You can only interact with a smart contract if it has public or external functions. The smart contract you've deployed has three external functions: `store`, `retrieve`, and `donate`.

From the end user's perspective, the function can be divided into reading functions and altering-state or writing functions. The smart contract has one reading function, `retrieve`.

Let's execute the `retrieve` method. Create a file named `execute_retrieve.py` and add the following code to it:

```
from web3 import Web3

with open("abi.json") as f:
    abi = f.read().strip()
```

```
w3 = Web3(Web3.IPCProvider('/tmp/geth.ipc'))

address = "0x8bb44f25E5b25ac14c8A9f5BFAcdFd1a700bA18B"
contract = w3.eth.contract(address=address, abi=abi)

number = contract.functions.retrieve().call()
print(number)
```

Make sure you change the address of the smart contract in the script to your smart contract's address. Also, change the location of the IPC file if necessary.

To interact with the smart contract, you don't need the bytecode anymore – you need the address. When you constructed a web3 object with the `w3.eth.contract` method, you provided the address and the ABI.

Then, you can call the `retrieve` method, like so:

```
contract.functions.retrieve().call()
```

The contract object comes from the `w3.eth.contract` method. Then, you received the `functions` property from the contract object. Afterward, you called the method from the smart contract. Since the `retrieve` method does not accept any argument, you call it like so: `retrieve()`. Note that if the method you want to call has arguments, you must provide them. Lastly, you executed the `call` method from the `retrieve` method.

When executing reading functions (functions that don't alter the state of the smart contract), you don't need to use accounts.

Run the script:

```
(.venv) $ python execute_retrieve.py
0
```

The `retrieve` methods display the value of the `num` state variable, which is `0`. Let's change the `num` state variable! It's time to create a script that will execute the `store` method, which will change the state of the smart contract. This time, you need an account.

Create a file named `execute_store.py` and add the following code to it:

```
from web3 import Web3

with open("abi.json") as f:
    abi = f.read().strip()

w3 = Web3(Web3.IPCProvider('/tmp/geth.ipc'))
```

```
account_address = "0xA7603e35744EA8d88F5F66f0dE7bcE24f2Bf8F2f"
with open("my_keystore/UTC--2023-07-08T18-06-12.161925018Z--a7603e3574
4ea8d88f5f66f0de7bce24f2bf8f2f") as keyfile:
    encrypted_key = keyfile.read()
    password = "mypassword123"
    account_private_key = w3.eth.account.decrypt(encrypted_key,
password)

address = "0x8bb44f25E5b25ac14c8A9f5BFAcdFd1a700bA18B"
contract = w3.eth.contract(address=address, abi=abi)
```

You should already be familiar with the code. Now, you can set up an account so that you can sign the transaction. You can also set up the contract object.

Make sure you change the account address to the address you created with `clef`. Change the IPC file's location if necessary, as well as the password. Change the value of the address variable to the `address` variable of the smart contract you deployed.

Then, add the following code at the bottom of the file:

```
transaction = contract.functions.store(99).build_transaction({
    'from': account_address,
    'nonce': w3.eth.get_transaction_count(account_address),
    'gas': 200000,
    'gasPrice': Web3.to_wei('50', 'gwei')
})

signed_transaction = w3.eth.account.sign_transaction(transaction,
account_private_key)

transaction_hash = w3.eth.send_raw_transaction(signed_transaction.
rawTransaction)

print(transaction_hash)
```

The way you executed the `store` method is similar to the `retrieve` method. You can get the `store` method from `contract.functions`. Because the `store` method accepts an argument, you have to provide the argument like this:

```
contract.functions.store(99)
```

However, instead of executing the `call` method, you used the `build_transaction` method to create a transaction. You must provide `gas` and `gasPrice`, among other things, to the `transaction` object. It's similar to when you build a transaction to deploy a smart contract.

The rest of the code should be familiar to you. You signed the transaction and sent the transaction.

Run the script:

```
(.venv) $ python execute_store.py
b"Y\x8b_J_#\x1c\xf0\x05F -L\x19\x11'\xbfP \xadFC\x9al\xdfH\xbb~\xe4\
xe1\xa9\x89"
```

How do you know it works? You can execute the script that executes the `retrieve` method again:

```
(.venv) $ python execute_retrieve.py
99
```

As you can see, the `num` state variable changed from `0` to `99`.

You can also send ETH when executing a payable function. The smart contract has one payable function, `donate`. Executing the `donate` function and sending some ETH along the way is similar to executing the `store` function. You only need to add another key to the `transaction` object.

Create the `execute_donate.py` file and add the following code to it:

```
from web3 import Web3

with open("abi.json") as f:
    abi = f.read().strip()

w3 = Web3(Web3.IPCProvider('/tmp/geth.ipc'))

account_address = "0xA7603e35744EA8d88F5F66f0dE7bcE24f2Bf8F2f"

with open("my_keystore/UTC--2023-07-08T18-06-12.161925018Z--a7603e3574
4ea8d88f5f66f0de7bce24f2bf8f2f") as keyfile:
    encrypted_key = keyfile.read()
    password = "mypassword123"
    account_private_key = w3.eth.account.decrypt(encrypted_key,
password)

address = "0x8bb44f25E5b25ac14c8A9f5BFAcdFd1a700bA18B"
contract = w3.eth.contract(address=address, abi=abi)
```

You should be familiar with this code. As usual, make sure you change the account address to the address you created with `clef`. Change the IPC file's location if necessary, as well as the password. Change the value of the `address` variable to the address of the smart contract you deployed.

Then, add the following code at the bottom of the file:

```
transaction = contract.functions.donate().build_transaction({
    'from': account_address,
    'value': Web3.to_wei('1', 'ether'),
```

```
    'nonce': w3.eth.get_transaction_count(account_address),
    'gas': 200000,
    'gasPrice': Web3.to_wei('50', 'gwei')
})

signed_transaction = w3.eth.account.sign_transaction(transaction,
account_private_key)

transaction_hash = w3.eth.send_raw_transaction(signed_transaction.
rawTransaction)

print(transaction_hash)
```

Let's explain the code. You executed the `donate` method like so:

```
contract.functions.donate()
```

In the `transaction` object inside the `build_transaction` method, notice that there is the `value` key that has a value of 1 ETH. This is how you send ETH when executing the payable function.

Run the script:

```
(.venv) $ python execute_donate.py
b'h\xf8\x8299\xfa\xe4\x10\x8e\xa7\xc7\x92\x97\xcb~\xdb\x8a\x9f\x0e;|K\
x1a\xc1\x06\x99\xb7 /d\xb4\xb9'
```

Also, remember that in the `donate` method in the smart contract, we log the event. Let's look at the definition of the `donate` method in the smart contract, which logs the event:

```
@external
@payable
def donate():
    log Donation(msg.sender, msg.value)
```

The `Donation` event declaration looks like this:

```
event Donation:
    donatur: indexed(address)
    amount: uint256
```

When you transfer ETH to the smart contract, you can also query the event with `web3.py`.

Create a script named `query_event.py` and add the following code to it:

```
from web3 import Web3

with open("abi.json") as f:
```

```
    abi = f.read().strip()

w3 = Web3(Web3.IPCProvider('/tmp/geth.ipc'))

address = "0x8bb44f25E5b25ac14c8A9f5BFAcdFd1a700bA18B"
contract = w3.eth.contract(address=address, abi=abi)

logs = contract.events.Donation().get_logs(fromBlock=0)

for log in logs:
    print(f"The donatur address is {log.args.donatur}")
    print(f"The donation amount is {log.args.amount}")
```

Make sure you change the IPC file's location if necessary. Replace the value of the `address` variable with your smart contract's address.

To get the events, you used the `events` property of the contract object. Then, you received the `Donation` event from the `events` property. From there, you executed the `get_logs` methods with the parameter that specified `fromBlock` with a value of `0`. Here, `fromBlock` indicates that we want to search the log event at the beginning. You can choose another value if you don't want to search for event from the first block in the blockchain.

Then, you can iterate the logs object to get information about the event. If you may have noticed, the `Donation` event has two properties: `donatur` and `amount`.

Run the script:

```
(.venv) $ python query_event.py
The donatur address is 0xA7603e35744EA8d88F5F66f0dE7bcE24f2Bf8F2f
```

The donation amount is `1000000000000000000`.

With that, you have successfully retrieved the log events with a Python script! You can now say that you know how to use `web3.py` to interact with smart contracts. Before we close this chapter, let's learn about some security tips.

Security consideration

You must have noticed that we hardcoded the password and the private key in Python scripts. For a tutorial, it's fine. But in production, this is not recommended.

To make this more secure, you can put the private key in an environment variable. To demonstrate this, create a file named `env_var.py` and add the following code to it:

```
import os

print(os.environ["PRIVATE_KEY"])
```

Then, run the script after setting the environment variable:

```
(.venv) $ export PRIVATE_KEY=0xverysecret
(.venv) $ python env_var.py
0xverysecret
```

You can also get the password interactively. Create a file named `get_password.py` and add the following code to it:

```
from getpass import getpass

password = getpass()
print(password)
```

Run the script:

```
(.venv) $ python get_password.py
Password:
mypassword123!
```

Of course, these are undoubtedly not the most secure means. You can use the hardware wallet to sign the transaction. Then, you can send the transaction through your computer. Alternatively, you could split a key into different keys, but that's beyond the scope of this book.

Summary

In this chapter, we learned how to write Python scripts to interact with smart contracts using the `web3.py` library. First, we installed the library. Then, we installed Ganache and Geth, the development blockchains. After, we wrote a Python script to deploy a smart contract that's written in Vyper. Next, we learned how to interact with smart contracts, from executing the function to read the state variable, executing the function to change the state variable, and executing the function to send ETH to the smart contract to querying event logs.

In the next chapter, we'll learn how to use a blockchain development framework to develop smart contracts and interact with them.

5
Ape Framework

In this chapter, we will learn how to use **Ape Framework** to develop smart contracts, exploring compiling smart contracts, deploying them to blockchain, and creating tests. Although you can do these things with the `web3.py` library and the Vyper compiler, Ape Framework gives you more tools and convenience to work on smart contracts. Using Ape Framework can boost your productivity. You don't have to set up a compiler and install libraries separately. Ape Framework can wrap long, complicated commands into simpler forms.

The following topics will be covered in this chapter:

- Installing Ape Framework
- Developing smart contracts
- Testing smart contracts
- Networks in blockchain

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_5`.

Installing Ape Framework

While you can use the `web3.py` library and the Vyper compiler to develop smart contracts, Ape Framework can make you more productive. To make an analogy, using the `web3.py` library and the Vyper compiler to develop smart contracts is like using the PHP programming language to develop web applications; there are PHP frameworks, such as Laravel or Symfony, that can help you develop web applications much faster.

The framework helps you structure projects. That way, you don't have to waste your time thinking about how to set up your projects. The framework also comes with the batteries to develop your projects. You don't have to install the libraries that you need one by one. Using the framework instills discipline in you when developing your projects.

Ape Framework is a Python application. So, you can install it inside a Python virtual environment. But first you need to install some dependencies in Ubuntu Linux: `$ sudo apt install build-essential autoconf libtool pkg-config python3-dev`.

After installing the dependencies, create a Python virtual environment.

```
$ python3.10 -m venv .venv
$ source .venv/bin/activate
$ (.venv) $ pip install --upgrade pip
```

Then, install the framework this way:

```
(.venv) $ pip install eth-ape'[recommended-plugins]'
```

The command not only installed Ape Framework but also some recommended plugins for Ape Framework. One of them is the Vyper compiler, which you will need.

However, if you want to install a specific version, you can run this command:

```
(.venv) $ pip install eth-ape'[recommended-plugins]'==0.7.23
```

At the time of writing, the latest version of Ape is 0.8. However, due to some issues with it, You should use Ape version 0.7. To make sure it's correctly installed, you can run the `ape` command:

```
(.venv) $ ape --version
0.7.23
```

You can also use Docker to use Ape Framework:

```
$ docker run apeworx/ape --version
0.8.3
```

As you can see, the latest version of Ape fetched with Docker is 0.8.3.

As you can see, the latest version of Ape was fetched with Docker is 0.8.3. Now, to view all the plugins you've installed, you can run this command:

```
(.venv) $ ape plugins list
Installed Plugins
  alchemy      0.7.3
  ens          0.7.1
  etherscan    0.7.0
```

```
  foundry      0.7.1
  hardhat      0.7.3
  infura       0.7.3
  solidity     0.7.1
  template     0.7.0
  tokens       0.7.1
  vyper        0.7.0
```

Now, to get started on creating a project with Ape Framework, you must create an empty directory first. Let's name the directory `simple_storage`:

```
$ mkdir simple_storage
$ cd simple_storage
```

Then you can initialize this directory to become the blockchain project with the `ape init` command:

```
(.venv) $ ape init
```

You'll then be asked to name the project:

```
Please enter project name:
```

Enter `Simple Storage` and then press *Enter*. You'll get a success message. The command and the output will look like this:

```
(.venv) $ ape init
Please enter project name: Simple Storage
SUCCESS: Simple Storage is written in ape-config.yaml
```

Since the command initialized the directory, the directory has within it some sub-directories. You can check it yourself by running this command:

```
$ ls -a
.  ..  ape-config.yaml  contracts  .gitignore  scripts  tests
```

Now that the project has been set up, it's time to develop smart contracts with Ape Framework.

Developing smart contracts

Before you develop smart contracts, let's understand the structure of the project. We start with the `ape-config.yaml` file. This is the configuration file of your project. Let's take a look at the content of the file:

```
$ cat ape-config.yaml
name: Simple Storage
```

Right now, it only contains the name of the project. But later, you'll add more configurations to this file for your project.

`.gitignore` is a configuration file that ignores files in this project directory when you commit the project into a Git repository.

The `contracts` directory is a directory that keeps your Vyper files. The `scripts` directory is where you put your Python files that will interact with smart contracts. The tests directory is the place of test files for smart contracts.

Let's develop a smart contract project using Ape Framework!

Compiling a smart contract

Create a Vyper file named `SimpleStorage.vy` inside the `contracts` folder and add the following code to it:

```
# @version ^0.3.0

num: uint256

@external
def store(num: uint256):
    self.num = num

@external
def retrieve() -> uint256:
    return self.num
```

Then, you can compile the Vyper file not by using the Vyper compiler directly, but by using Ape Framework. Run the `ape compile` command:

```
(.venv) $ ape compile
INFO: Compiling 'SimpleStorage.vy'.
```

Behind the scenes, the `ape` command is still using the Vyper compiler. The compilation result is kept in `.build/SimpleStorage.json`. You can find the ABI and the bytecode in the JSON file. But you will also find many other things, such as the abstract syntax tree and the source map of the smart contract.

Because the file is big, it's useful to read the content of the JSON file with the `jq` tool. But you need to install it first.

```
$ sudo apt install jq
```

This will allow you to read the JSON file with `jq`. Run the following command to get the ABI of the smart contract:

```
$ jq '.abi' .build/SimpleStorage.json
[
  {
    "inputs": [
      {
        "name": "num",
        "type": "uint256"
      }
    ],
    "name": "store",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
  },
  {
    "inputs": [],
    "name": "retrieve",
    "outputs": [
      {
        "name": "",
        "type": "uint256"
      }
    ],
    "stateMutability": "nonpayable",
    "type": "function"
  }
]
```

You can get the bytecode too:

```
$ jq '.deploymentBytecode.bytecode' .build/SimpleStorage.json
"0x61004f6100186300000000396100 4f600001630000000 0f3600436101561000d576
10044565b60003560e01c3461004a57636057361d811861002b57600435600055005b6
32e64cec1811861004257600054604052602060 40f35b505b60006000fd5b600080fd"
```

Of course, you can also open the JSON file with the text editor and read it yourself.

Now that you have compiled your smart contract, the next step is to deploy the smart contract to blockchain.

Deploying the smart contract

Let's deploy the smart contract's bytecode to blockchain. For this, you can use the **Geth development blockchain**. Make sure the Geth development blockchain runs in `http://127.0.0.1/8545`. Create an account with `clef` and add some ETH from the first account.

Refer to the previous chapter to learn how to install and run the Geth development blockchain and bootstrap some ETH for an account. Don't use Ganache because Ganache has some bugs where you cannot deploy smart contracts compiled with Vyper version 0.3.10.

For convenience, to run the Geth development blockchain, run the following command:

```
$ geth --dev --http --http.api eth,web3,net
```

Then you can create an account with `clef`:

```
$ mkdir my_keystore
$ clef newaccount --keystore my_keystore
```

To boostrap ETH to the new account, use the following:

```
$ geth attach /tmp/geth.ipc
```

Replace `/tmp/geth.ipc` with your IPC file's path. Inside the Geth console, send some ETH from the first account:

```
> eth.sendTransaction({from: eth.accounts[0], to:
"0x62a34D301a137b465Fb8a87Cb115293378c2BF89", value: web3.toWei(10,
"ether")})
```

Replace `0x62a34D301a137b465Fb8a87Cb115293378c2BF89` with the account address that you created with `clef`.

To deploy the smart contract, you need to create a deployment script inside the `scripts` folder. Create a file named `deploy.py` and add the following code to it:

```
from ape import accounts, project
import os


def main():
    password = os.environ["MY_PASSWORD"]
    dev = accounts.load("dev")
    dev.set_autosign(True, passphrase=password)
    contract = project.SimpleStorage.deploy(sender=dev)
    num_value = contract.retrieve.call()
    print(f"The num value is {num_value}")
```

In this `deployment` script, you retrieve the password from an environment variable that locks your private key. Then you load the account using the `dev` alias. You have to unlock this account so the script can use the private key. Then you deploy the smart contract to blockchain. After that, you interact with the smart contract by executing the `retrieve` method. Finally, you need to put the things you want to do inside a function named `main` in this script so Ape Framework can use it.

Before you execute the script, you need to have an account with the `dev` alias. You can import one of the accounts in Ganache to Ape Framework. You can now prepare the private key. Then, you can refer to the previous chapter to see how to get the private key from any account in Ganache.

Run this command to import the account from Ganache:

```
(.venv) $ ape accounts import dev
Enter Private Key:
```

The command, as you can see here, tried to import an account using the `dev` alias. Paste the private key that you get from any account you create in the Geth development blockchain. Then, you will be asked for a password to protect this private key:

```
Create Passphrase to encrypt account:
```

Once you have created the password, you will be asked for confirmation:

```
Repeat for confirmation:
```

The command and the output to import the account from Ganache should look like this:

```
$ ape accounts import dev
Enter Private Key:
Create Passphrase to encrypt account:
Repeat for confirmation:
SUCCESS: A new account '0xe7121100683622eD7399E11F0e6aA98e9a17251a'
has been added with the id 'dev'
```

Your account could be different from the account displayed here.

The account is kept in `the .ape` folder inside the home directory. You can check it with this command:

```
$ cat ~/.ape/accounts/dev.json
{"address": "e7121100683622ed7399e11f0e6aa98e9a17251a",
"crypto": {"cipher": "aes-128-ctr", "cipherparams":
{"iv": "e6ae7e7e79c2ed476dea947fee12ca38"}, "ciphertext":
"ce7f160f9bf32d4da019bb3c8e233ffc9da115f5f16cb3f198a269593eb75387",
"kdf": "scrypt", "kdfparams": {"dklen": 32, "n": 262144, "r":
1, "p": 8, "salt": "36110f4b3415d5cd733d6d42bd567bdc"}, "mac":
"88f1bf3fc34015b1f93ae840d30530254622d366f6a2cf21635e1fdc0c89c0e7"},
"id": "a11ecdd2-f4f9-4fac-a4d2-d02176d2359f", "version": 3}
```

This file, as you can see, contains your private key encrypted with your password.

You can also generate a new account with Ape Framework. Run the following command to create a new account:

```
(.venv) $ ape accounts generate dev2
Add extra entropy for key generation...:
```

The command creates an account using the `dev2` alias. It asks you to add extra entropy. Type some random characters and then press *Enter*.

Then, you'll be asked whether you want a mnemonic that is displayed or not:

```
Show mnemonic? [Y/n]:
```

Just answer `Y` because you want to see the mnemonic. Then, you'll get a password prompt.

```
Create Passphrase to encrypt account:
```

Once you have typed in your password, you will be asked for confirmation:

```
Repeat for confirmation:
```

The command and the output should look like this:

```
(.venv) $ ape accounts generate dev2
Add extra entropy for key generation...:
Show mnemonic? [Y/n]: Y
INFO: Newly generated mnemonic is: praise nut spend enlist promote
fall toy similar silk spell visual foster
Create Passphrase to encrypt account:
Repeat for confirmation:
SUCCESS: A new account '0x076f13BB1Db39F982cd8Dcc01DC829e45064f163'
with HDPath m/44'/60'/0'/0/0 has been added with the id 'dev2'
```

The mnemonic and account will be different because every time you generate a new account, it will create a totally different account.

> **Mnemonic**
>
> A mnemonic consists of 12 or 24 English words (but you can choose another language). It is similar to a private key. This mnemonic holds a key to your account. If your mnemonic is leaked to other people, then your account will be compromised. With one mnemonic, you can generate many accounts with different private keys. People prefer to keep mnemonics instead of private keys because it's more natural and easier to remember.

Then, you need to modify the `ape-config.yaml` file so you can deploy the smart contract to the Geth development blockchain. Add the following configuration to the file:

```
geth:
  ethereum:
    local:
      uri: http://127.0.0.1:8545
```

Make sure the URI matches the URL of your Geth.

Set your password to the environment variable:

```
$ export MY_PASSWORD=yourpassword
```

Then, you can execute the deployment script:

```
(.venv) $ ape run deploy --network ethereum:local:geth
WARNING: Connecting Geth plugin to non-Geth client 'Ganache'.
WARNING: Danger! This account will now sign any transaction it's
given.
WARNING: Using cached key for dev
INFO: Confirmed
0xcafa7217acb93967d633b7ef9345d1695a384364b6c1251e89aaf9459712c40d
(total fees paid = 82872555534604)
SUCCESS: Contract 'SimpleStorage' deployed to:
0x5b2f58f567Abb595e2DD83478B4EAC1FaDB1762D
The num value is 0
```

The deployment address that you get will be different, but the `num` value should be `0`.

Notice that you have successfully run the Python deployment script with the `ape run` command. You didn't run it with the `python` command. You passed the name of the script, `deploy`, without the `.py` extension. Then you added the `network` flag to deploy the smart contract to the local blockchain.

Now, let's change the `num` value. Create another script named `update_num.py` inside the `scripts` directory and add the following code to it:

```
from ape import accounts, project
import os


def main():
    password = os.environ["MY_PASSWORD"]
    address = os.environ["SIMPLE_STORAGE_ADDRESS"]
    dev = accounts.load("dev")
    dev.set_autosign(True, passphrase=password)
    contract = project.SimpleStorage.at(address)
    contract.store(19, sender=dev)
    print("Updating num value")
```

```
    num_value = contract.retrieve.call()
    print(f"The num value is {num_value}")
```

The code shown here is similar to the first snippet. This time, you don't deploy the smart contract anymore because you want to use the deployed smart contract on Ganache. So, you load the smart contract with the `at` method. To execute the transaction that updates the state of the smart contract, you use the method name, which is the `store` method from the contract object. But you have to add the `sender` keyword as the argument to the `store` method.

Before executing the script you must set two environment variables. The first one is the password of the key file and the second one is the address of the smart contract. You can find the address from the output of the first script:

```
(.venv) $ export SIMPLE_STORAGE_
ADDRESS=0x5b2f58f567Abb595e2DD83478B4EAC1FaDB1762D
(.venv) $ export MY_PASSWORD=yourpassword
(.venv) $ ape run update_num --network ethereum:local:geth
WARNING: Connecting Geth plugin to non-Geth client 'Ganache'.
WARNING: Danger! This account will now sign any transaction it's
given.
WARNING: Using cached key for dev
INFO: Confirmed
0x002aea6dc419c200acd10a6efbc0331f3cb37335799ecd1c4e3d1a9185de5cce
(total fees paid = 29937038379468)
Updating num value
The num value is 19
```

Here as well, you run the script with the `ape run` command. Then, you provide the name of the script without the `.py` extension. You also use the `ethereum:local:geth` network argument.

As you can see, the script has changed the value of the `num` variable successfully.

Now that you have compiled, deployed, and interacted with a smart contract, it's time to create unit tests for the smart contract.

Testing smart contracts

It's true that you can test the smart contract manually like you did before by using scripts, but that's not efficient. It's better to create unit tests to test your smart contract. It helps you develop smart contracts better. The project that you work on has a `tests` folder. You can put unit tests there.

Create a file named `test_simple_storage.py` inside the `tests` folder and add the following code to it:

```
import pytest


@pytest.fixture
def deployer(accounts):
    return accounts[0]

@pytest.fixture
def contract(deployer, project):
    return deployer.deploy(project.SimpleStorage)
def test_retrieve(contract):
    num = contract.retrieve.call()
    assert num == 0

def test_store(contract, deployer):
    contract.store(19, sender=deployer)
    num = contract.retrieve.call()
    assert num == 19
```

In this test, you created a fixture of the `deployer` account with the `deployer` function and the `@pytest.fixture` annotation. The function accepts the `accounts` argument, which contains all accounts in the test. These accounts are loaded with ETH.

Then you created another fixture of the smart contract with the `contract` function and the `@pytest.fixture` annotation. The function accepts the `deployer` argument and the `project` argument. The `deployer` argument comes from the `deployer` function you defined previously. The project is the fixture provided by Ape Framework. If you can recall, you used the `project` object in the scripts you've written. Inside the contract function, you deployed the contract of `SimpleStorage`.

Then comes the unit tests of the methods you defined in the smart contract. The first test is `test_retrieve`, which tests the `retrieve` method. You provided the `contract` argument. This argument comes from the `contract` function you defined previously. Inside the method, you call the `retrieve` method with the `call` method.

The last test is `test_store`, which tests the `store` method. This test function has another argument, which is the `deployer` argument. You need it because when you call the `store` method, you need to fill in the `sender` keyword argument.

You can run the test with the `ape test` command:

```
(.venv) $ ape test
============================================================================
test session starts ========================================================
```

```
====================
platform linux -- Python 3.10.10, pytest-7.4.0, pluggy-1.2.0
rootdir: /home/arjuna/Code/books/Hands-On-Blockchain-for-Python-
Developers--2nd-Edition/chapter_5/simple_storage
plugins: eth-ape-0.6.13, web3-6.6.1
collected 2 items

tests/test_simple_storage.py
..
                                                          [100%]

============================================================================
==
2 passed in 0.60s ==================================================
==
==================
```

As you can see, the two tests passed successfully. If you only want to run one test, for example, the `test_store` function, you can run the test like this:

```
(.venv) $ ape test tests/test_simple_storage.py -k test_store
============================================================================
test session starts ================================================
====================
platform linux -- Python 3.10.10, pytest-7.4.0, pluggy-1.2.0
rootdir: /home/arjuna/Code/books/Hands-On-Blockchain-for-Python-
Developers--2nd-Edition/chapter_5/simple_storage
plugins: eth-ape-0.6.13, web3-6.6.1
collected 2 items / 1 deselected / 1 selected

tests/test_simple_storage.py .

               [100%]

============================================================== 1
passed, 1 deselected in 0.43s ======================================
============================
```

Now, we can place the test fixtures inside the test file. That's fine. But you can also put it in another file in case you want to reuse the fixtures in another test file. Ape Framework has a dedicated place for you to add the fixtures and other global variables. Create the `conftest.py` file inside the `tests` folder and add the following code to it:

```
import pytest


@pytest.fixture
```

```
def deployer(accounts):
    return accounts[0]

@pytest.fixture
def contract(deployer, project):
    return deployer.deploy(project.SimpleStorage)
```

Then you can remove these fixtures from `tests/test_simple_storage.py`. The file should look like this:

```
def test_retrieve(contract):
    num = contract.retrieve.call()
    assert num == 0

def test_store(contract, deployer):
    contract.store(19, sender=deployer)
    num = contract.retrieve.call()
    assert num == 19
```

If you run the `ape test` command again, the result will be the same:

```
(.venv) $ ape test
============================================================================
test session starts ========================================================
====================
platform linux -- Python 3.10.10, pytest-7.4.0, pluggy-1.2.0
rootdir: /home/arjuna/Code/books/Hands-On-Blockchain-for-Python-
Developers--2nd-Edition/chapter_5/simple_storage
plugins: eth-ape-0.6.13, web3-6.6.1
collected 2 items

tests/test_simple_storage.py ..

                                                                [100%]

============================================================================
= 2 passed in 0.61s ========================================================
====================
```

You have thus completed the smart contract software development process from end to end. But you've only used the smart contract on development blockchains such as Ganache and Geth. It's time now to explore other blockchains as well.

Networks in blockchain

In the course of this chapter, you have learned how to develop smart contracts on development blockchains. You've tested contracts manually with scripts and you've tested them automatically with unit tests. Now, it's time to deploy a smart contract to the Ethereum mainnet. Before you do that, you should familiarize yourself with other kinds of blockchains that are helpful for developing smart contracts: testnet blockchains. The most popular Ethereum testnet blockchain is **Sepolia**.

> **What are the differences between Sepolia and Ganache?**
>
> Ganache only runs on your computer and only one computer maintains it. Sepolia runs on the internet and is used by many people. There is a set of validators maintaining the Sepolia network.

This means you can deploy your smart contract to Sepolia and let your friends in a different country use it. It's not that you cannot do that with Ganache, but Ganache is not scalable for that purpose. It's only designed to be used by one person.

In Sepolia, you don't need to set up a validator node in order to use the network. You can use someone else's node to access the network. This is easier.

There are companies that provide these kinds of services. The two most popular companies known to provide such services are Infura and Alchemy. In this chapter, you'll use Alchemy. Head to their website at `https://www.alchemy.com/`, sign up, and then create an app at `https://dashboard.alchemy.com/apps`. You will then see this page:

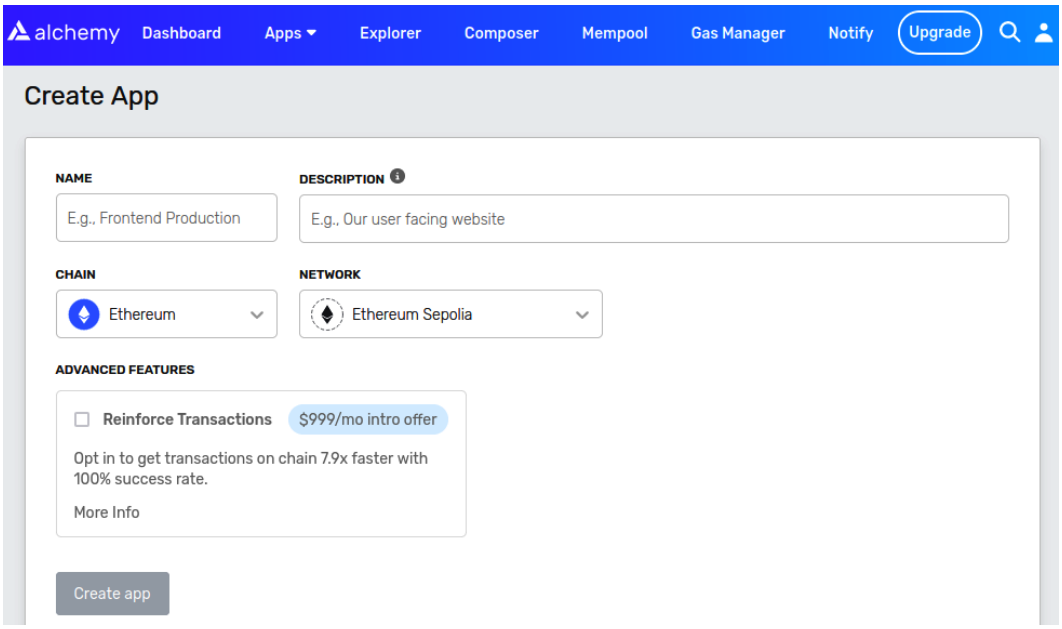


Figure 5.1: The Create App page

Fill in the name. Choose **Ethereum** on the **CHAIN** field. Then, choose **Ethereum Sepolia** in the **NETWORK** field. The app will show up in the **Apps** section as shown in the next figure:

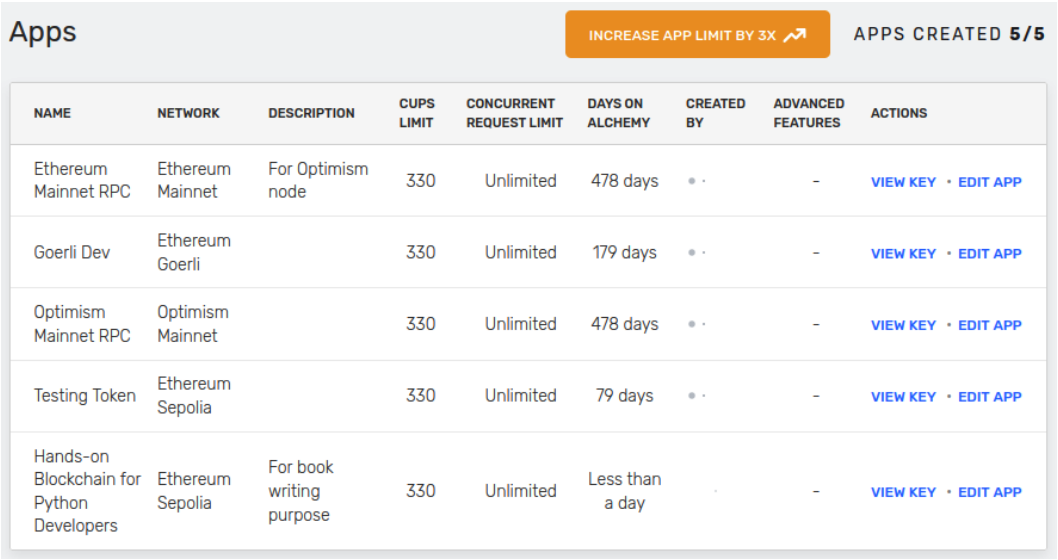


Figure 5.2: The Apps section

Click the **VIEW KEY** button for your newly created app. You'll be presented with a pop-up window that shows your API key as shown in the next figure:

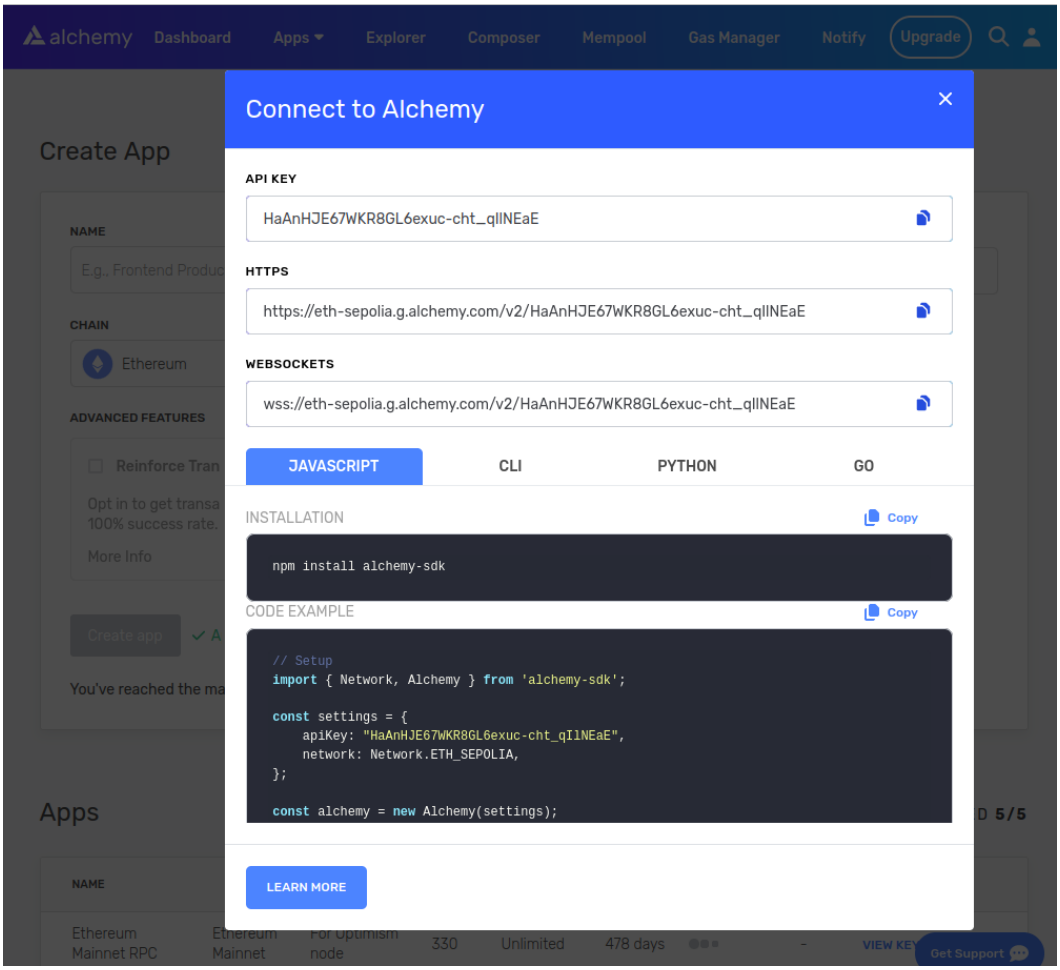


Figure 5.3: The API key page

Copy the API key from this window. Ape Framework needs it once you have connected to the Sepolia network.

Now, head back to your project. But before you are able to deploy the smart contract to the Sepolia network, you need to fill your account with ETH in the Sepolia network. In Ganache, you get 10 accounts that are filled with ETH. In the Sepolia network, you need to find them from faucets. A faucet is a website where you can ask for a bit of ETH for your account. It's free ETH, but the amount of ETH that you can ask for in a period of time is limited.

Go to `https://sepoliafaucet.com/` to get ETH. You need to log in using your Alchemy account. For each period of time, you can get 0.5 Sepolia ETH. That's enough for your requirements in this case. When you visit the URL, you'll see a web page that looks as follows:

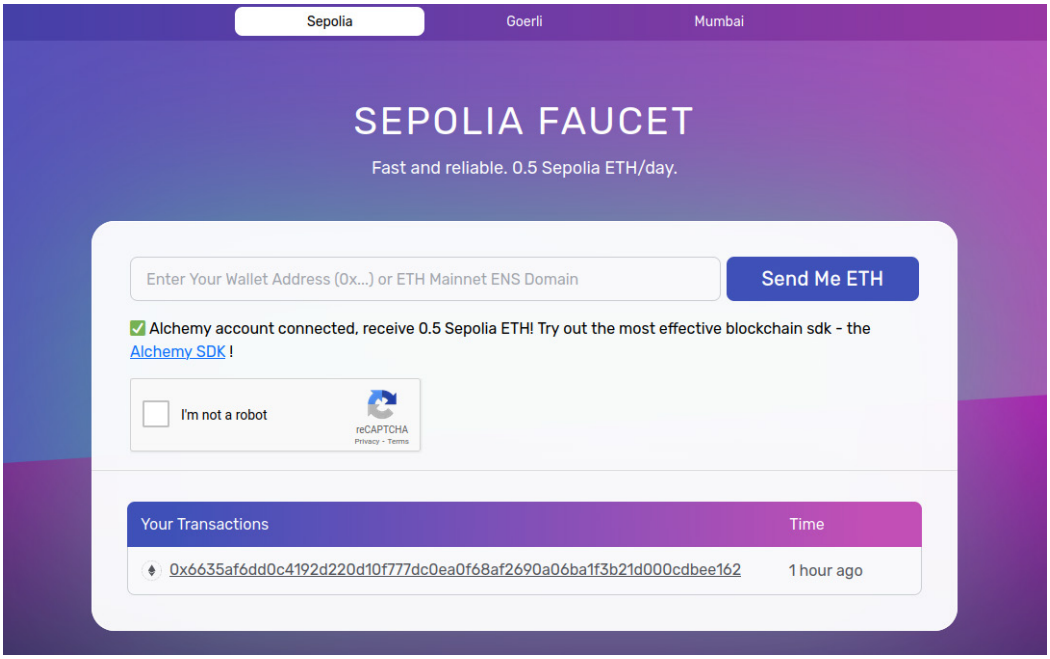


Figure 5.4: The faucet page

Fill in the address of your account. Then, click the **Send Me ETH** button.

There are other faucet websites that you can use, such as: `https://cloud.google.com/application/web3/faucet/ethereum/sepolia`, `https://www.infura.io/faucet/sepolia`, and `https://faucet.quicknode.com/drip`. After this, head back to your project. Set this environment variable with your API key from Alchemy:

```
$ export WEB3_ETHEREUM_SEPOLIA_ALCHEMY_API_KEY=yourAPIkeyfromAlchemy
```

Then you can run your deployment script but with a different value for the `network` flag:

```
(.venv) $ ape run deploy --network ethereum:sepolia:alchemy
WARNING: Danger! This account will now sign any transaction it's
given.
WARNING: Using cached key for dev
INFO: Submitted https://sepolia.etherscan.io/
tx/0xdd6b8f8932d4fb978f2b142e48bec22a3dc23a5ec5054c06388781b1232703d5
Confirmations (2/2): 100%|
|  2/2
```

```
[00:29<00:00, 14.81s/it]
INFO: Confirmed 0xdd6b8f8932d4fb978f2b142e48bec22a3dc23a5ec5054c063887
81b1232703d5 (total fees paid = 70209993540680)
SUCCESS: Contract 'SimpleStorage' deployed to: 0x3EF724BfbFa6558a0897d
1E3760D66C806e85deF
The num value is 0
```

The key is to use a different value for the `network` flag, which is `ethereum:sepolia:alchemy`. If you have decided to use a different company to connect to Sepolia, such as Infura, then you can use this value for the `network` flag, `ethereum:sepolia:infura`, but you need to set a particular environment variable, which is `WEB3_ETHEREUM_SEPOLIA_INFURA_API_KEY`.

Now that you've deployed your smart contract to the Sepolia network, you can check the transaction here: `https://sepolia.etherscan.io/tx/0xdd6b8f8932d4fb978f2b142e48bec22a3dc23a5ec5054c06388781b1232703d5`. The contract's address is `0x3EF724BfbFa6558a0897d1E3760D66C806e85deF`, but yours may be different. You need to tell your friend the address of this smart contract and the blockchain you use, which is the Sepolia network, before they can use your smart contract.

If you want to deploy the smart contract to the Ethereum mainnet, the process is similar. You would need to use a different value for the `network` flag, which would be either `ethereum:mainnet:alchemy` or `ethereum:mainnet:infura`. But you need to get real ETH. You cannot ask for this ETH for free. You can buy ETH from cryptocurrency exchanges such as Coinbase or Binance.

Summary

In this chapter, we learned how to develop a smart contract using Ape Framework. First, we installed the framework. Then, we compiled a Vyper file. After this, we created a script to deploy and interact with a smart contract. To make the smart contract well-tested, we wrote unit tests. Later, we used a different blockchain, Sepolia, to deploy our smart contract. In the next chapter, we'll develop a practical smart contract. This application will have the benefits of the smart contract platform, such as transparency and security from cheating.

6

Building a Practical Decentralized Application

In this chapter, we will learn how to build a practical decentralized application. We've learned the bolts and nuts of the Vyper language. We've used the Ape Framework to build and deploy smart contracts onto the blockchain. Now, it's time to write a program on the blockchain. Then, we'll see the values in the decentralized program that we're going to build.

The application that we're going to build is a voting application. Many people will find this application useful for finding solutions to many issues. The nature of voting requires the application to be secure from tampering. We'll see why the importance of the decentralized nature of the application is paramount.

The following topics will be covered in this chapter:

- Writing the voting smart contract
- Writing the unit test of the voting smart contract
- Adding the delegation to the voting smart contract
- The values of the decentralized nature of the voting smart contract

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_6`.

Writing the voting smart contract

The application you're going to build is a voting application. Users will vote for some proposals. In this example, you can imagine students choosing where to have a study tour for their class. Do they choose to spend it on a beach or a mountain?

You'll build the voting decentralized application using the Ape framework that you learned in the previous chapter. So, the flow of the development should be familiar to you.

First, create a Python virtual environment:

```
$ python3.10 -m venv .venv
$ source .venv/bin/activate
$ (.venv) $ pip install --upgrade pip
```

Then you can install the Ape Framework:

```
(.venv) $ pip install eth-ape'[recommended-plugins]'==0.7.23
```

After installing the framework, you can start developing the application by creating the directory for the voting application:

```
(.venv) $ mkdir voting_app
(.venv) $ cd voting_app
```

Inside the directory, you should initialize the project using the `ape` command:

```
(.venv) $ ape init
Please enter project name:
```

The most suitable name for the project is `Voting App`. After filling in the name of the project, hit *Enter*. You should get the following output:

```
(.venv) $ ape init
Please enter project name: Voting App
SUCCESS: Voting App is written in ape-config.yaml
```

Create a Vyper file named `VotingApp.vy` inside the `contracts` folder.

First, you add the required compiler version to it:

```
# @version ^0.3.0
```

To encapsulate a vote, add the `Voter` struct to the code:

```
struct Voter:
    weight: uint256
    voted: bool
    vote: uint256
```

The `weight` attribute represents the heaviness of the vote. You can set this property to 1 if you want everyone to have an equal voice. But there can also be situations where some people have heavier votes – maybe you want to give more weight to votes from people with children, for example.

The `voted` attribute makes sure that a user doesn't vote more than once. Once a user votes, they cannot make another vote.

The `vote` attribute points to the proposal a user chooses. As an example, in a poll of preferred locations, the value `0` could represent the beach and the value `1` could represent the mountain.

After writing the data structure for the voting functionality, you write the data structure for the proposal or something that is being voted on. Add the `Proposal struct` to the code:

```
struct Proposal:
    name: String[100]
    voteCount: uint25
```

The `name` attribute is the name of the proposal. In this example, it's a *beach* or a *mountain*. The `voteCount` attribute represents how many votes a proposal has.

After creating two data structures, it's time to create state variables to hold the state of the voting application. Add the following state variables to the code:

```
voters: public(HashMap[address, Voter])
proposals: public(HashMap[uint256, Proposal])
voterCount: public(uint256)
chairperson: public(address)
amountProposals: public(uint256)
```

The `voters` state variable holds the users who vote in this voting application. The key of the `HashMap` is the address. Remember that an address in Ethereum represents an account in Ethereum or a blockchain user in Ethereum. The value of the `HashMap` is the `Voter` struct or a `vote`.

The `proposals` state variable represents the integer to the `Proposal` struct. A user or voter will choose an integer later. This integer is mapped to the `Proposal` struct in this `HashMap`.

The `voterCount` state variable keeps track of the number of voters in this voting application.

The `chairperson` state variable is the user that's going to allow users to become voters. Not every user can vote in this smart contract. The `chairperson` needs to allow them first.

The `amountProposals` state variable refers to the number of proposals a user can vote on.

Because of a limitation in Vyper, you need the next line:

```
MAX_NUM_PROPOSALS: constant(uint256) = 3
```

Ideally, you don't need to hardcode the number of the maximum proposals in the smart contract. But in Vyper, when you want to iterate the proposals you add to the smart contract, you cannot use the dynamic variable. You have to use a literal value or a constant that's not going to change. You'll see why you need this constant variable later in the code. For now, let's move on.

After declaring all the state variables you need, now is the time to create a function to initialize the smart contract by adding the `__init__` function to the code:

```
@external
def __init__():
    self.chairperson = msg.sender
```

Here, you initialized the `chairperson` state variable with the user who deploys this smart contract.

Then you need a function to add a `proposal` to the smart contract. You add the following function to the code:

```
@external
def addProposal(_proposalName: String[100]):
    assert msg.sender == self.chairperson
    i: uint256 = self.amountProposals
    self.proposals[i] = Proposal({
        name: _proposalName,
        voteCount: 0
    })
    self.amountProposals += 1
```

In the `addProposal` function, you make sure only the `chairperson` user is able to call this function. Then you add the `Proposal` struct to the `proposals` `HashMap`. Lastly, you increase the `amountProposals` state variable.

After that, you need to add a function so the `chairperson` can allow only those users who can vote:

```
@external
def giveRightToVote(voter: address, _weight: uint256):
    assert msg.sender == self.chairperson
    assert not self.voters[voter].voted
    assert self.voters[voter].weight == 0
    self.voters[voter].weight = _weight
    self.voterCount += 1
```

In the function, you make sure only the `chairperson` can call this function. The function accepts an address or a user, then the weight of the vote. You then make sure the added user has not voted yet. Then you set the weight to the vote weight of the user. Lastly, you increase the `voterCount` state variable. So, if you want 10 users to participate in this voting application, the chairperson needs to call this method 10 times. To allow a user to vote, you need to give a value bigger than 0 to the `weight` argument.

The next function you should add is the function to vote:

```
@external
def vote(proposal: uint256):
    assert not self.voters[msg.sender].voted
    assert proposal < self.amountProposals

    self.voters[msg.sender].vote = proposal
    self.voters[msg.sender].voted = True

    self.proposals[proposal].voteCount += self.voters[msg.sender].
weight
    self.voters[msg.sender].weight = 0
```

This function can be called by anyone as long as the user has not voted yet. The argument to this function is a proposal of the integer type. You can look at the `proposals` `HashMap` state variable to view the name of the proposal. Technically speaking, casting a vote means using the weight of a user's vote to increase the `voteCount` of the `Proposal` struct inside the `proposals` state variable. Then, of course, you need to set the weight back to zero and the `voted` variable to `true` in the voter data to prevent another vote being cast. You also set how the user voted on the proposal to retain the voting history. *So to reiterate, voting means giving the weight of the vote to the votes a proposal has. If a user's vote has the weight 3 and the user votes for proposal A, the proposal A will have 3 votes.*

Then you need to add a function to count the votes to find out the winning proposal:

```
@view
@internal
def _winningProposal() -> uint256:
    winning_vote_count: uint256 = 0
    winning_proposal: uint256 = 0
    for i in range(MAX_NUM_PROPOSALS):
        if self.proposals[i].voteCount > winning_vote_count:
            winning_vote_count = self.proposals[i].voteCount
            winning_proposal = i
    return winning_proposal
```

The function is an internal function, so only other functions in the smart contract can call this function. An external user cannot call this function. You iterate all proposals to count how many votes they have. You set the winning proposal to the `winning_proposal` variable. Notice that you use the `MAX_NUM_PROPOSALS` constant in the iteration. You may think that you should use the `amountProposals` state variable in the iteration. But Vyper cannot accept dynamic variables such as `amountProposals`. The variable's value can change anytime and in the iteration, you need to use a number that's not going to change. If you want to add 10 proposals, then you need to change the value of `MAX_NUM_PROPOSALS` before deploying it onto the blockchain.

To use this function, you need to add other functions that can be called by external users:

```
@view
@external
def winningProposal() -> uint256:
    return self._winningProposal()

@view
@external
def winnerName() -> String[100]:
    return self.proposals[self._winningProposal()].name
```

The difference between these two functions is that the first one gives the winning proposal as an integer, while the latter gives the name of the winning proposal in characters.

The smart contract is now complete. Now you can compile it by running the `ape` command:

```
(.venv) $ ape compile
```

Now, let's write a unit test to make sure the smart contact works as intended.

Writing the unit test of the voting smart contract

To make sure the smart contract works well, you need to confirm it with unit tests. The way you write the unit test is similar to what you did in the previous chapter.

First, create the `conftest.py` file inside the `tests` folder, then add the following code to it:

```
import pytest


@pytest.fixture
def deployer(accounts):
    return accounts[0]

@pytest.fixture
def contract(deployer, project):
    return deployer.deploy(project.VotingApp)
```

This file contains the initial setup and configuration required for your test file to run properly. The `deployer` function returns the deployer, which uses the first test account. The `contract` function is used to deploy the voting smart contract.

Next, let's create the `test_voting_app.py` file inside the `tests` folder and add the following code to it:

```
import pytest
from ape.exceptions import ContractLogicError
def test_chairperson(contract, deployer):
    chairperson = contract.chairperson()
    assert chairperson == deployer
```

This function makes sure that the chairperson is the deployer of the smart contract. In other words, this test tests the `__init__` function of the smart contract.

Let's add a test to test the `addProposal` function:

```
def test_addProposal(contract, deployer):
    assert contract.amountProposals() == 0
    assert contract.proposals(0).name == ""
    contract.addProposal("beach", sender=deployer)
    assert contract.amountProposals() == 1
    assert contract.proposals(0).name == "beach"
```

This tests that, after calling the `addProposal` function, the number of proposals increases and the name of the proposal added can be retrieved.

You also want to test the fail case, making sure another user cannot call this function:

```
def test_addProposal_fail(contract, accounts):
    with pytest.raises(ContractLogicError):
        contract.addProposal("beach", sender=accounts[1])
```

You use another user, `accounts[1]`, to call the `addProposal` function. In the `test` function, you catch the exception.

Then let's move to the next function you want to test, `giveRightToVote`. Add the `test_giveRightToVote` function to the code:

```
def test_giveRightToVote(contract, deployer, accounts):
    user = accounts[1]
    assert contract.voterCount() == 0
    assert contract.voters(user).weight == 0
    contract.giveRightToVote(user, 1, sender=deployer)
    assert contract.voterCount() == 1
    assert contract.voters(user).weight == 1

    power_user = accounts[2]
    assert contract.voters(power_user).weight == 0
```

```
    contract.giveRightToVote(power_user, 9, sender=deployer)
    assert contract.voterCount() == 2
    assert contract.voters(power_user).weight == 9
```

To test this function, you check the weight of a voter before and after calling the `giveRightToVote` function. The weight of the voter should increase by the `weight` argument you send into the `giveRightToVote` function.

You also test the `voterCount` method in this test. Every time you successfully call `giveRightToVote`, the `voterCount` method will give the increased number.

Then, of course, you want to test the fail case of the `giveRightToVote` function:

```
def test_giveRightToVote_fail(contract, deployer, accounts):
    user = accounts[1]
    with pytest.raises(ContractLogicError):
        contract.giveRightToVote(user, 1, sender=accounts[1])
```

This unit test tests the `giveRightToVote` function as well, only its fail case. The test calls the `giveRightToVote` function with another user. As you know, only the chairperson is able to call this function.

Next, test the `vote` function by adding the following code to the test file:

```
def test_vote(contract, deployer, accounts):
    contract.addProposal("beach", sender=deployer)
    contract.addProposal("mountain", sender=deployer)
    user = accounts[1]
    user2 = accounts[2]
    contract.giveRightToVote(user, 1, sender=deployer)
    contract.giveRightToVote(user2, 1, sender=deployer)

    assert contract.voters(user).weight == 1
    assert contract.voters(user).voted == False
    assert contract.voters(user).vote == 0
    assert contract.proposals(0).voteCount == 0
    contract.vote(0, sender=user)
    assert contract.proposals(0).voteCount == 1
    assert contract.voters(user).weight == 0
    assert contract.voters(user).voted == True
    assert contract.voters(user).vote == 0

    assert contract.voters(user2).weight == 1
    assert contract.voters(user2).voted == False
    assert contract.voters(user2).vote == 0
```

```
    assert contract.proposals(1).voteCount == 0
    contract.vote(1, sender=user2)
    assert contract.proposals(1).voteCount == 1
    assert contract.voters(user2).weight == 0
    assert contract.voters(user2).voted == True
    assert contract.voters(user2).vote == 1
```

First, in this preceding test function, you set up the two proposals, `beach` and `mountain`, to be voted on. Then you give the right to vote to two users. The first user votes for the first proposal. The second user votes for the second proposal. After voting, you check that the vote action from each user has added points to the correct proposal. You also make sure the `weight` value of the voters is set to zero. Lastly, you want to make sure the `voted` is set to `True` to make sure the voters cannot vote again.

You also want to test the fail case of the `vote` function by adding the following test to the test file:

```
def test_vote_fail(contract, deployer, accounts):
    contract.addProposal("beach", sender=deployer)
    contract.addProposal("mountain", sender=deployer)
    user = accounts[1]
    contract.giveRightToVote(user, 1, sender=deployer)

    contract.vote(0, sender=user)
    with pytest.raises(ContractLogicError):
        contract.vote(0, sender=user)
```

In this test case, you try to make a user vote twice. That's not allowed in the smart contract. The second voting action should throw an exception, which is caught by the test.

The last test is to test the `winnerName` function. You need to make sure that the function to return the winner of the voting works correctly. Let's add the following test to the code:

```
def test_winnerName(contract, deployer, accounts):
    contract.addProposal("beach", sender=deployer)
    contract.addProposal("mountain", sender=deployer)
    user1 = accounts[1]
    user2 = accounts[2]
    user3 = accounts[3]
    contract.giveRightToVote(user1, 1, sender=deployer)
    contract.giveRightToVote(user2, 1, sender=deployer)
    contract.giveRightToVote(user3, 1, sender=deployer)

    contract.vote(0, sender=user1)
    contract.vote(1, sender=user2)
    contract.vote(1, sender=user3)
```

```
    assert contract.proposals(0).voteCount == 1
    assert contract.proposals(1).voteCount == 2
    assert contract.winnerName() == "mountain"
```

In this test, you create two proposals. Then you give the right to vote to three users. Two users choose proposal 1, `mountain`. One user chooses proposal 0, `beach`. When you call the `winnerName` function, you will get the name of the winning proposal, `mountain`.

You can run the test using `pytest`:

```
(.venv) $ py.test tests/test_voting_app.py
============================= test session starts
==============================
platform linux -- Python 3.10.10, pytest-7.4.0, pluggy-1.2.0
rootdir: /home/arjuna/Code/books/Hands-On-Blockchain-for-Python-
Developers--2nd-Edition/chapter_6/voting_app
plugins: eth-ape-0.6.15, web3-6.8.0
collected 8 items

tests/test_voting_app.py
........                                                [100%]

============================== 8 passed in 3.23s
==============================
```

With the tests passing, you can be pretty sure that the voting application runs as intended. If you want, you can deploy the smart contract onto the blockchain and invite your friends to vote on some proposals.

But what if some of your friends are just too busy and want to delegate their decisions to other people, such as their neighbors or colleagues? For this, we can add some delegation functionality to the smart contract.

Adding delegation to the voting smart contract

Most voting smart contracts that you encounter on the blockchain have the delegation feature. A lot of smart contracts have a treasury or money accumulated on the smart contract. How to spend this money is decided by the governance process in the smart contract. The governance process is basically the users of the smart contract voting on issues including where to spend this money.

Imagine there are a thousand users. Not everyone is keen on voting for every issue of the smart contract. What will happen is that most users will delegate their voting power to other people, who are small in number. These people will vote for most issues on the smart contract and are actively involved in the governance process.

To create a voting smart contract with the delegation feature, you need to add some functions that transfer the weight of a vote from one user to another.

Let us create a new file named `DelegateVotingApp.vy` inside the `contracts` folder and add the following code to it:

```
# @version ^0.3.0
struct Voter:
    weight: uint256
    voted: bool
    delegate: address
    vote: uint256
```

You used the same `struct` but this time it has another property, which is `delegate`. This property refers to the account that gets the voting power from another user. If a user doesn't delegate to other people, then the `delegate` property of this `struct` will always be zero address.

Then we add the following additional code to the file:

```
struct Proposal:
    name: String[100]
    voteCount: uint256

voters: public(HashMap[address, Voter])
proposals: public(HashMap[uint256, Proposal])
voterCount: public(uint256)
chairperson: public(address)
amountProposals: public(uint256)

MAX_NUM_PROPOSALS: constant(uint256) = 3
```

We discussed this code in the previous section. Let's next add methods to check whether a user has delegated their voting power or not:

```
@view
@internal
def _delegated(addr: address) -> bool:
    return self.voters[addr].delegate != empty(address)

@view
@external
def delegated(addr: address) -> bool:
    return self._delegated(addr)
```

As you can see in the preceding functions, you check whether the user has delegated their voting power by probing the value of the `delegate` property of the `Vote` struct.

You also want to know how a user voted for a proposal. Does the user vote directly (without delegation) or through delegation (someone votes for them)? Let's add the functions to do that:

```
@view
@internal
def _directlyVoted(addr: address) -> bool:
    return self.voters[addr].voted and (self.voters[addr].delegate ==
empty(address))

@view
@external
def directlyVoted(addr: address) -> bool:
    return self._directlyVoted(addr)
```

In the preceding functions, to vote directly means the `voted` property is true and the `delegate` property is false.

Then we add the functions to initialize the smart contract, add a proposal, and give the right to vote:

```
@external
def __init__():
    self.chairperson = msg.sender

@external
def addProposal(_proposalName: String[100]):
    assert msg.sender == self.chairperson
    i: uint256 = self.amountProposals
    self.proposals[i] = Proposal({
        name: _proposalName,
        voteCount: 0
    })
    self.amountProposals += 1

@external
def giveRightToVote(voter: address, _weight: uint256):
    assert msg.sender == self.chairperson
    assert not self.voters[voter].voted
    assert self.voters[voter].weight == 0
    self.voters[voter].weight = _weight
    self.voterCount += 1
```

The preceding functions exist in the original voting smart contract. Let's move on by adding functions to give the voting power:

```
@internal
def _forwardWeight(delegate_with_weight_to_forward: address):
    assert self._delegated(delegate_with_weight_to_forward)
    assert self.voters[delegate_with_weight_to_forward].weight > 0
    target: address = self.voters[delegate_with_weight_to_forward].
delegate
    for i in range(4):
        if self._delegated(target):
            target = self.voters[target].delegate
        else:
            break

    weight_to_forward: uint256 = self.voters[delegate_with_weight_to_
forward].weight
    self.voters[delegate_with_weight_to_forward].weight = 0
    self.voters[target].weight += weight_to_forward

    if self._directlyVoted(target):
        self.proposals[self.voters[target].vote].voteCount += weight_
to_forward
        self.voters[target].weight = 0
```

First, you make sure the delegated address has been chosen by the user and the weight of the user's vote should be greater than 0.

Then you check whether the given delegated address has delegated the voting power to another user! There could be a case where user A delegates to user B, but user B already delegated to user C, and finally, user C has delegated to user D. In this case, you want to transfer the voting power of user A to user D. But generally, it is wise to limit the iteration of passing on delegations to four times, otherwise, the operation becomes too expensive on the blockchain.

Then you empty the weight of the vote of the user who delegates their vote and put their voting weight into the delegated address voting weight.

If the delegated user has voted already, then you also empty the weight of the user after increasing the vote count of the proposal that the delegated user voted on.

The function is an internal function, meaning you cannot call it directly. So, you need to create an external function to utilize this function:

```
@external
def delegate(to: address):
    assert not self.voters[msg.sender].voted
```

```
    assert to != msg.sender
    assert to != empty(address)

    self.voters[msg.sender].voted = True
    self.voters[msg.sender].delegate = to

    self._forwardWeight(msg.sender)
```

Before calling the `_forwardWeight` function, you set the `voted` property of the voter to be true, then set the `delegate` property to the delegated address. Of course, you make sure the user hasn't voted yet and that the delegated address is not the user themselves, or empty.

Then you have the `vote` function, which comes from the original voting smart contract:

```
@external
def vote(proposal: uint256):
    assert not self.voters[msg.sender].voted
    assert proposal < self.amountProposals

    self.voters[msg.sender].vote = proposal
    self.voters[msg.sender].voted = True

    self.proposals[proposal].voteCount += self.voters[msg.sender].
weight
    self.voters[msg.sender].weight = 0
```

Finally, you should add the functions to get the winning proposal:

```
@view
@internal
def _winningProposal() -> uint256:
    winning_vote_count: uint256 = 0
    winning_proposal: uint256 = 0
    for i in range(MAX_NUM_PROPOSALS):
        if self.proposals[i].voteCount > winning_vote_count:
            winning_vote_count = self.proposals[i].voteCount
            winning_proposal = i
    return winning_proposal

@view
@external
def winningProposal() -> uint256:
    return self._winningProposal()

@view
```

```
@external
def winnerName() -> String[100]:
    return self.proposals[self._winningProposal()].name
```

These functions are a function to vote and the functions to get the winning proposals.

Now the smart contract is complete, you can write the unit tests. This time, you just write tests for the delegation features. Create the `test_delegate_voting_app.py` file inside the `tests` folder and add the following code to it:

```
import pytest
from ape.exceptions import ContractLogicError

def test_delegate(delegate_contract, deployer, accounts):
    user = accounts[1]
    user2 = accounts[2]
    delegate_contract.giveRightToVote(user, 1, sender=deployer)
    delegate_contract.giveRightToVote(user2, 2, sender=deployer)

    assert delegate_contract.voters(user).weight == 1
    assert delegate_contract.voters(user).voted == False
    assert delegate_contract.voters(user).vote == 0
    assert delegate_contract.voters(user2).weight == 2
    assert delegate_contract.voters(user2).voted == False
    assert delegate_contract.voters(user2).vote == 0
    delegate_contract.delegate(user, sender=user2)
    assert delegate_contract.voters(user).weight == 3
    assert delegate_contract.voters(user).voted == False
    assert delegate_contract.voters(user).vote == 0
    assert delegate_contract.voters(user2).weight == 0
    assert delegate_contract.voters(user2).voted == True
    assert delegate_contract.voters(user2).vote == 0
```

The preceding test function tests whether, after delegating, a user loses their vote weight and the `voted` property becomes `true`.

Let's add another test:

```
def test_delegate_2_levels(delegate_contract, deployer, accounts):
    user = accounts[1]
    user2 = accounts[2]
    user3 = accounts[3]
    delegate_contract.giveRightToVote(user, 1, sender=deployer)
    delegate_contract.giveRightToVote(user2, 2, sender=deployer)
    delegate_contract.giveRightToVote(user3, 5, sender=deployer)
```

```
    assert delegate_contract.voters(user).weight == 1
    assert delegate_contract.voters(user).voted == False
    assert delegate_contract.voters(user).vote == 0
    assert delegate_contract.voters(user2).weight == 2
    assert delegate_contract.voters(user2).voted == False
    assert delegate_contract.voters(user2).vote == 0
    assert delegate_contract.voters(user3).weight == 5
    assert delegate_contract.voters(user3).voted == False
    assert delegate_contract.voters(user3).vote == 0
    delegate_contract.delegate(user, sender=user2)
    delegate_contract.delegate(user2, sender=user3)
    assert delegate_contract.voters(user).weight == 8
    assert delegate_contract.voters(user).voted == False
    assert delegate_contract.voters(user).vote == 0
    assert delegate_contract.voters(user2).weight == 0
    assert delegate_contract.voters(user2).voted == True
    assert delegate_contract.voters(user2).vote == 0
    assert delegate_contract.voters(user3).weight == 0
    assert delegate_contract.voters(user3).voted == True
    assert delegate_contract.voters(user3).vote == 0
```

In this test function, `user2` delegates to `user` then `user3` delegates to `user2`. In this case, the vote weight from `user3` moves to `user`.

Next, add another test to test the voting process after delegating two levels:

```
def test_vote_after_delegate_2_levels(delegate_contract, deployer,
accounts):
    delegate_contract.addProposal("beach", sender=deployer)
    delegate_contract.addProposal("mountain", sender=deployer)
    user = accounts[1]
    user2 = accounts[2]
    user3 = accounts[3]
    delegate_contract.giveRightToVote(user, 1, sender=deployer)
    delegate_contract.giveRightToVote(user2, 2, sender=deployer)
    delegate_contract.giveRightToVote(user3, 5, sender=deployer)
    delegate_contract.delegate(user, sender=user2)
    delegate_contract.delegate(user2, sender=user3)
    delegate_contract.vote(1, sender=user)
    assert delegate_contract.voters(user).weight == 0
    assert delegate_contract.voters(user).voted == True
    assert delegate_contract.voters(user).vote == 1
    assert delegate_contract.winnerName() == "mountain"
```

```
    assert delegate_contract.proposals(0).voteCount == 0
    assert delegate_contract.proposals(1).voteCount == 8
```

This is similar to the previous test function, but you test the `vote` function after delegating two levels and make sure the correct proposal wins.

The last test is for the delegation process after the delegated user has voted. First, you set up the necessary conditions:

```
def test_delegate_after_vote(delegate_contract, deployer, accounts):
    delegate_contract.addProposal("beach", sender=deployer)
    delegate_contract.addProposal("mountain", sender=deployer)
    user = accounts[1]
    user2 = accounts[2]
    user3 = accounts[3]
    delegate_contract.giveRightToVote(user, 1, sender=deployer)
    delegate_contract.giveRightToVote(user2, 2, sender=deployer)
    delegate_contract.giveRightToVote(user3, 5, sender=deployer)
```

Then you add the test code on the bottom of the test function:

```
    delegate_contract.vote(1, sender=user)
    delegate_contract.delegate(user, sender=user2)
    delegate_contract.delegate(user2, sender=user3)

    assert delegate_contract.voters(user2).weight == 0
    assert delegate_contract.voters(user2).voted == True
    assert delegate_contract.voters(user2).vote == 0
    assert delegate_contract.voters(user2).delegate == user
    assert delegate_contract.voters(user3).weight == 0
    assert delegate_contract.voters(user3).voted == True
    assert delegate_contract.voters(user2).vote == 0
    assert delegate_contract.voters(user3).delegate == user2

    assert delegate_contract.voters(user).weight == 0
    assert delegate_contract.voters(user).voted == True
    assert delegate_contract.voters(user).vote == 1
    assert delegate_contract.voters(user).delegate == "0x0000000000000
0000000000000000000000000"

    assert delegate_contract.winnerName() == "mountain"
    assert delegate_contract.proposals(0).voteCount == 0
    assert delegate_contract.proposals(1).voteCount == 8
```

In the preceding test, the target user voted first, then other users delegated to that user. The test makes sure the vote count of the chosen proposal increases.

Don't forget to modify the `conftest.py` file to include the new voting smart contract. Add the following fixture inside `conftest.py`:

```
@pytest.fixture
def delegate_contract(deployer, project):
    return deployer.deploy(project.DelegateVotingApp)
Then finally you can compile the new voting smart contract with the
delegation feature and run the unit test:
(.venv) $ ape compile
(.venv) $ py.test tests/test_delegate_voting_app.py
====================================== test session starts ============
==========================
platform linux -- Python 3.10.10, pytest-7.4.0, pluggy-1.2.0
rootdir: /home/arjuna/Code/books/Hands-On-Blockchain-for-Python-
Developers--2nd-Edition/chapter_6/voting_app
plugins: eth-ape-0.6.15, web3-6.8.0
collected 12 items

tests/test_delegate_voting_app.py
............                                                [100%]

======================================= 12 passed in 6.48s ============
==========================
```

Now that you have implemented voting smart contracts, let's contemplate what makes the blockchain voting application interesting.

The benefits of the decentralized nature of a voting smart contract

This voting application is a decentralized application. What does this mean? It means this smart contract, once deployed, cannot be shut down because in a smart contract, there is no function to shut down the smart contract.

Compare this to a web voting application where, once deployed, the app can be shut down by the owner of the platform. This is a risk factor of centralization.

If you notice, there is a gatekeeper in the smart contract, called the *chairperson*. If a user wants to vote, they must be registered by the chairperson. Of course, you could remove the chairperson from the smart contract, meaning every address on the Ethereum blockchain could vote. This may lead to a Sybil attack where a lot of attackers could swamp the voting application to produce a specific output. Creating a new account on Ethereum doesn't cost much. You could create 1 million or 1 billion accounts and vote for a specific proposal. The chairperson's purpose is to prevent that possibility.

It's true that the chairperson can choose only certain users and thus bias the voting toward a certain proposal. *In this context, we assume the chairperson is impartial and neutral.*

Another benefit that emerges from the decentralized voting application is that the voting is transparent. When a user votes for a proposal, everyone can see the vote on the blockchain because the vote is recorded. Once a vote is cast, not even the chairman can undo the voting process. The chairman can swamp the voting result by giving the right to vote to other users who will vote for a certain proposal, but the chairman cannot change the votes of users.

Compare this to a voting application, such as polling on Twitter. The system administrator or people behind Twitter could theoretically change the voting result without anyone finding out by changing the votes of users.

Voting is crucial in a democracy. If the voting process can be tampered with, then people will lose their trust in democracy because it becomes impossible to reach a consensus.

While the voting application on the blockchain cannot prevent cheating 100%, at least the transparency of the blockchain can mitigate the damage of the cheating being done on the voting application. Suppose the chairperson chooses to give rights to vote to certain users and doesn't give rights to vote to rightful users – this action can be tracked easily.

To put it in a nutshell, decentralized applications are not totally safe from cheating but they have properties that can be used to make cheating harder.

Now that you've understood the values that come from the decentralized voting application, let's see how to improve our voting application.

Ideas for improvements

You can add some improvements to your voting application. While transparency is nice, sometimes you might prefer secrecy. You want to make sure the vote is safe from cheating, but not allow other people to find out how a certain user voted, at least until the proposal closes. This is called **shielded voting**.

Voting is very crucial in blockchain because it's the foundation of the decentralized nature of blockchain. People build many projects using not a top-down approach, but a bottom-up approach, such as allowing people to vote to decide how to spend the money that accumulates on a smart contract. This gives rise to **Decentralized Autonomous Organizations** (**DAOs**).

So a smart contract, or more likely a collection of smart contracts, can be an organization. It can be configured, for example, that in order for a function inside a smart contract to be executed, it requires an approval rate of 2/3 of the total number of users of the smart contract. Usually, functions that deal with money, such as money withdrawal, require approval from at least a majority of users.

Improvements don't have to be complicated. You can also add a feature to limit certain users from voting, either using an allowlist or a denylist. The allowlist or denylist could be in the form of `HashMap` and you would have a function to add a user to these lists. Then in the `vote` function, you would check whether the user is on the list or not (depending on whether it is an allowlist or a denylist).

Summary

In this chapter, we learned to develop a practical decentralized application for a useful smart contract. We built a voting application with which a user can vote for a specific proposal, using the Ape Framework. First, we wrote the smart contract. Then we created unit tests to make sure the voting smart contract works as intended. Finally, we covered the benefits of decentralized voting applications, what makes them different from traditional voting applications, and future improvements we could add to the voting application. In the next chapter, we'll build a front-end application for a smart contract.

Part 3: Graphical User Interface Applications

This section dives into the realm of building user-friendly interfaces for decentralized applications. You'll learn how to develop a graphical user interface (GUI) for your decentralized applications using the `PySide` library, enabling seamless interaction between end-users and your smart contracts. Additionally, this section will guide you through the process of creating a user-friendly cryptocurrency wallet application, one of the most important tools in the crypto ecosystem.

This section has the following chapters:

- *Chapter 7*, *Front-End Decentralized Application*
- *Chapter 8*, *Cryptocurrency Wallet*

7
Front-End Decentralized Application

In this chapter, we will learn how to write a front-end decentralized application using a Python **Graphical User Interface** (**GUI**) library. In the previous chapter, you wrote the voting smart contract using Vyper. However, using the voting smart contract with scripts is not convenient. We will, thus, build the GUI application on top of this smart contract to make the application easier to use.

The following topics will be covered in this chapter:

- Using the voting smart contract with scripts
- Installing the PySide library
- Creating the GUI application
- Connecting the GUI application with the smart contract

Technical requirements

To follow this chapter, you need one of the more recent versions of Linux and Python – no older than 3.8. You could try it on other platforms, such as Mac or Windows, but Linux is recommended. You could also run Linux with virtualization software such as VirtualBox. We will use Ubuntu Linux 22.04 LTS and Python 3.10. This chapter's code is based on the previous chapter's code. The full code is available at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_7`.

Using the voting smart contract with scripts

You developed the voting smart contract in the previous chapter. But you only wrote the smart contract and the unit test. You haven't yet deployed the smart contract onto the blockchain. To use the voting smart contract, you need to deploy it first.

Compiling the smart contract

Let's set up the Python environment and install the necessary libraries:

```
$ python3.10 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install --upgrade pip
(.venv) $ pip install eth-ape'[recommended-plugins]'==0.7.23
```

Then, create a project directory:

```
(.venv) $ mkdir voting_app
(.venv) $ cd voting_app
```

As usual, you initialize the project directory with the `ape` command and give the name `VotingApp` to the project:

```
(.venv) $ ape init
Please enter project name: VotingApp
SUCCESS: VotingApp is written in ape-config.yaml
```

Then, you should create `VotingApp.vy` inside the `contracts` folder. You should use the same content of `VotingApp.vy` from the previous chapter. You can get the code of `VotingApp.vy` from this URL: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_6/voting_app/contracts/VotingApp.vy`.

After the code is ready, you can compile the smart contract:

```
(.venv) $ ape compile
```

Before you run the smart contract, you need some accounts to use the smart contract. So, let's create some accounts.

Creating accounts

For this voting application usage, let's create four separate accounts using the `ape` command. You can use usernames such as `voter1`, `voter2`, `voter3`, and `chairperson`.

To generate the accounts with those usernames, you can use the `ape accounts` command as shown here:

```
(.venv) $ ape accounts generate voter1
Add extra entropy for key generation...:
Show mnemonic? [Y/n]: Y
```

```
INFO: Newly generated mnemonic is: raccoon charge broken floor bright
noise know cause evolve skate address fox
Create Passphrase to encrypt account:
Repeat for confirmation:
SUCCESS: A new account '0xd8DF097c9B1c901fa8fF32b8cBCf641fE5E35055'
with HDPath m/44'/60'/0'/0/0 has been added with the id 'voter1'
```

You'll get a different output because of randomness. Of course, you should not use the account above in production.

Then, for the usernames `voter2` and `voter3`, you create the accounts as well:

```
(.venv) $ ape accounts generate voter2
(.venv) $ ape accounts generate voter3
(.venv) $ ape accounts generate chairperson
```

As a reminder, your accounts are saved in `~/.ape/accounts/voter1.json`, `~/.ape/accounts/voter2.json`, `~/.ape/accounts/voter3.json`, and `~/.ape/accounts/chairperson.json`, where ~ is your user home directory.

Then, it's time to deploy the voting smart contract onto the blockchain. This time, you'll use the Geth development blockchain.

Setting up the Geth development blockchain

How to install Geth was explained in *Chapter 4*. As a reminder, you can run the Geth development blockchain like this:

```
$ geth --dev --http --http.api eth,web3,net
...
INFO [09-10|15:09:24.392] IPC endpoint opened
url=/tmp/geth.ipc
INFO [09-10|15:09:24.393] Generated ephemeral JWT secret
secret=0x580934dc6202e1967a16bd2b65c8db5d2e07201f8c8598696c03677b4d
a93b7e
INFO [09-10|15:09:24.394] HTTP server started
endpoint=127.0.0.1:8545 auth=false prefix= cors= vhosts=localhost
```

To get the `coinbase` address that holds a lot of ETH, connect to the Geth console:

```
$ geth attach /tmp/geth.ipc
Welcome to the Geth JavaScript console!

instance: Geth/v1.11.3-stable-5ed08c47/linux-amd64/go1.20.2
coinbase: 0x2e460cbc53ec240c2c39197f58c8f45df9b036a1
at block: 0 (Thu Jan 01 1970 07:00:00 GMT+0700 (WIB))
datadir:
```

```
modules: admin:1.0 clique:1.0 debug:1.0 engine:1.0 eth:1.0 miner:1.0
net:1.0 rpc:1.0 txpool:1.0 web3:1.0

To exit, press ctrl-d or type exit
>
```

The `coinbase` address from the preceding output is `0x2e460cbc53ec240c2c39197f58c8f45df9b036a1` but yours would be different.

This `coinbase` address has a lot of ETH:

```
> web3.fromWei(eth.getBalance(eth.accounts[0]))
1.157920892373161954235709850086879078532699846656405640394575840079131296
39927e+59
```

You want the `coinbase` account to send some ETH to the four accounts you created previously. To get the address of the account, you can display the content of the account file:

```
$ cat ~/.ape/accounts/voter1.json
{"address": "d8df097c9b1c901fa8ff32b8cbcf641fe5e35055",
"crypto": {"cipher": "aes-128-ctr", "cipherparams":
{"iv": "ed02728cd1be49dee4d982c79e104cb0"}, "ciphertext":
"4cb99721e91333cb3392f33901081848fafbab9bc4c2688936cc339633c5e038",
"kdf": "scrypt", "kdfparams": {"dklen": 32, "n": 262144, "r":
1, "p": 8, "salt": "f40f8240d6f24a6b4c9f7decea27009b"}, "mac":
"df57154c6e5af79edd848fb8086d54c274ba2f138c648b38e7308ac47ba3f1ac"},
"id": "d85a4817-2e9e-46ec-a8d6-f41f45746c64", "version": 3}
```

Look at the value of the address field, `"d8df097c9b1c901fa8ff32b8cbcf641fe5e35055"`.

Then, you go back to the Geth console and transfer some ETH from the `coinbase` account to this address:

```
> eth.sendTransaction({from: eth.accounts[0], to:
"0xd8df097c9b1c901fa8ff32b8cbcf641fe5e35055", value: web3.toWei(10,
"ether")})
"0x95c1c3a81d049050f317c3085577dd69ecfcc5de6340185b748e56c47cd1a94f"
> eth.sendTransaction({from: eth.accounts[0], to:
"0x05b480862bf5875a1d5dd277b67cc6e687053413", value: web3.toWei(10,
"ether")})
"0xe02222b7c1f8c93bc7df3f858cf53918f6887972a511a22bb5ab4b83fdf5c4cb"
> eth.sendTransaction({from: eth.accounts[0], to:
"0xa407e46cd32e50b264d2d07f77bb5a2436dbb1f0", value: web3.toWei(10,
"ether")})
"0x7f83a30e7511d1c5c073bed42a8409c43001f229409c3d75249f13d48c862d9d"
> eth.sendTransaction({from: eth.accounts[0], to:
"0xB39f66E1F22c2d673EB8F32887E49e79Cf29b264", value: web3.toWei(10,
"ether")})
"0xaf11cfe7212b1052e71fdd5fce893421d23b5e2db2f3e65261d637ead532a975"
```

Each account now has 10 ETH in this development blockchain.

After this, you need to adjust the project configuration file to use the Geth development blockchain. You can edit `ape-config.yml` and replace everything with the following content:

```
name: VotingApp

geth:
  ethereum:
    local:
      uri: http://127.0.0.1:8545
```

Now that the accounts are loaded with ETH, you are ready to deploy the smart contract onto blockchain.

Deploying the smart contract onto blockchain

To deploy the smart contract, you can create a script named `deploy.py` inside the `scripts` directory and add the following code to it:

```
from ape import accounts, project
import os


def main():
    password = os.environ["CHAIRPERSON_PASSWORD"]
    chairperson = accounts.load("chairperson")
    chairperson.set_autosign(True, passphrase=password)
    contract = project.VotingApp.deploy(sender=chairperson)
    chairperson = contract.chairperson()
    print(f"The chairperson account is {chairperson}")
```

Before you run the deployment script, you must set the password to the `CHAIRPERSON_PASSWORD` environment variable:

```
(.venv) $ export CHAIRPERSON_PASSWORD=youraccountpassword
```

Then you can deploy the smart contract to the Geth development blockchain:

```
(.venv) $ ape run deploy --network ethereum:local:geth
INFO: Connecting to existing Geth node at http://127.0.0.1:8545/
[hidden].
WARNING: Danger! This account will now sign any transaction it's
given.
WARNING: Using cached key for chairperson
INFO: Confirmed
0x2c55a5299d90ac3e6fb73cf008263bf3840b91f3bea81b8577d7251d4ea749de
(total fees paid = 124910276288409)
```

```
SUCCESS: Contract 'VotingApp' deployed to:
0x2402B2Dc4a3F8A5aA144297a073DE8F26F97d204
The chairperson account is 0xB39f66E1F22c2d673EB8F32887E49e79Cf29b264
```

From the preceding output, you can see that the smart contract has been deployed to `0x2402B2Dc4a3F8A5aA144297a073DE8F26F97d204`. Yours would be different.

Save this address in the environment variable:

```
(.venv) $ export VOTING_APP_
ADDRESS=0x2402B2Dc4a3F8A5aA144297a073DE8F26F97d204
```

The smart contract has been deployed and you can use it now.

Creating and running the scripts

Before voting, you need to have some proposals and the chairperson needs to give the right to vote to the voters.

So, let's create a script to add proposals to the voting smart contract as a chairperson. Name the script `add_proposals.py` and put it inside the `scripts` folder. Add the following code to the script:

```
from ape import accounts, project
import os


def main():
    password = os.environ["CHAIRPERSON_PASSWORD"]
    address = os.environ["VOTING_APP_ADDRESS"]
    chairperson = accounts.load("chairperson")
    chairperson.set_autosign(True, passphrase=password)
    contract = project.VotingApp.at(address)
    contract.addProposal("beach", sender=chairperson)
    contract.addProposal("mountain", sender=chairperson)
```

The preceding code loads the smart contract object using the address you saved to the environment variable previously. Then you call the `addProposal` method twice using the chairperson account.

You can execute the script using this command:

```
(.venv) $ ape run add_proposals --network ethereum:local:geth
INFO: Connecting to existing Geth node at http://127.0.0.1:8545/
[hidden].
WARNING: Danger! This account will now sign any transaction it's
given.
WARNING: There are no token lists installed
```

```
ValueError: Default token list has not been set.
(Use `--verbosity DEBUG` to see full stack-trace)
WARNING: Using cached key for chairperson
INFO: Confirmed
0xb99ac0771a5567e605e9777694020a54d2cd8af07444aa2f6ed527ee061befa8
(total fees paid = 32675774984985)
WARNING: Using cached key for chairperson
INFO: Confirmed
0xa3223c7e98a48417d618d1e46fc27fcd36018603a3bd5b9ce49a45abeb24ff9f
(total fees paid = 23383488476502)
```

So, this will give you two proposals.

Now, let's give a right to vote using the script. Create a script named `give_right_to_vote.py` and put it inside the `scripts` directory. Then you add the following code to it:

```
from ape import accounts, project
import os


def main():
    password = os.environ["CHAIRPERSON_PASSWORD"]
    address = os.environ["VOTING_APP_ADDRESS"]
    voter_1 = os.environ["VOTER_1_ADDRESS"]
    voter_2 = os.environ["VOTER_2_ADDRESS"]
    voter_3 = os.environ["VOTER_3_ADDRESS"]
    chairperson = accounts.load("chairperson")
    chairperson.set_autosign(True, passphrase=password)
    contract = project.VotingApp.at(address)
    contract.giveRightToVote(voter_1, 1, sender=chairperson)
    contract.giveRightToVote(voter_2, 1, sender=chairperson)
    contract.giveRightToVote(voter_3, 1, sender=chairperson)
```

The script is similar to the previous script. Once you get the smart contract object, you execute the `giveRightToVote` method three times, each for a different voter.

Before executing the script, you must set the environment variables with the voters' accounts first:

```
(.venv) $ export VOTER_1_
ADDRESS=0xd8df097c9b1c901fa8ff32b8cbcf641fe5e35055
(.venv) $ export VOTER_2_
ADDRESS=0x05b480862bf5875a1d5dd277b67cc6e687053413
(.venv) $ export VOTER_3_
ADDRESS=0xa407e46cd32e50b264d2d07f77bb5a2436dbb1f0
```

Then you can run the script:

```
(.venv) $ ape run give_right_to_vote --network ethereum:local:geth
INFO: Connecting to existing Geth node at http://127.0.0.1:8545/
[hidden].
WARNING: Danger! This account will now sign any transaction it's
given.
WARNING: Using cached key for chairperson
INFO: Confirmed
0x6547ff7619567fc3cbb0b45888122d703643bea418b2cdace47f048ff9146bdc
(total fees paid = 19082690603860)
WARNING: Using cached key for chairperson
INFO: Confirmed
0x86337d00dc1382f0c2f11a33b1c425c431f20d2248426f94f8ad2e28e8c69ce0
(total fees paid = 12665936653634)
WARNING: Using cached key for chairperson
INFO: Confirmed
0xb007a9c6a1976ba1dd840b6538399f146df45405c53e04f11e45ea4945fa6481
(total fees paid = 11097225299456)
```

After the preparation, the voters can vote for the proposals living in the smart contract.

So, it's time to create a script that can be run by a voter, not a chairperson. Create a script named `vote.py` and put it inside the `scripts` folder, then add the following code to it:

```
from ape import accounts, project
import os


def main():
    password = os.environ["VOTER_PASSWORD"]
    address = os.environ["VOTING_APP_ADDRESS"]
    voter_account = os.environ["VOTER_ACCOUNT"]
    proposal = os.environ["PROPOSAL"]
    voter = accounts.load(voter_account)
    voter.set_autosign(True, passphrase=password)
    contract = project.VotingApp.at(address)
    contract.vote(proposal, sender=voter)
```

In this case, you execute the `vote` method using a voter's account. You can set the voter account using the environment variable. You can also set the voter's password and the proposal choice using the environment variables.

So, you must set some environment variables first:

```
(.venv) $ export VOTER_PASSWORD=youraccountpassword
(.venv) $ export VOTER_ACCOUNT=voter1
(.venv) $ export PROPOSAL=0
```

Then, you can run the script as a voter:

```
(.venv) $ ape run vote --network ethereum:local:geth
INFO: Connecting to existing Geth node at http://127.0.0.1:8545/
[hidden].
WARNING: Danger! This account will now sign any transaction it's
given.
WARNING: Using cached key for voter1
INFO: Confirmed
0x730150d06bb0d4e02f9b1f0ea4f87e36990f5e7724e6a2826b8412698b7f9f42
(total fees paid = 12864078639808)
```

So, the `voter1` account has given a vote for proposal `0` or `"beach"`.

You can use the same script to vote as a different voter, but of course, you need to set different values for some environment variables.

So, you've used the voting smart contract using the scripts. As you can see, it's quite cumbersome to use. Why not create a friendly GUI application? Let's create a voting desktop application so users can vote easily.

Installing the PySide library

To create a desktop application, you must build it with a GUI library. There are a lot of GUI libraries and some Python GUI libraries. For this application, you're going to use the `PySide` library. You can install `PySide` using `pip`:

```
(.venv) $ pip install pyside6
```

To test the library, you can write a simple GUI application. Let's create a script named `hello_pyside.py` and add the following code to it:

```
import sys
from PySide6 import QtCore, QtWidgets


class TextWidget(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()

        self.label = QtWidgets.QLabel("Hello Pyside")
        self.button = QtWidgets.QPushButton("Button")

        self.layout = QtWidgets.QVBoxLayout(self)
        self.layout.addWidget(self.label)
        self.layout.addWidget(self.button)
```

```
if __name__ == "__main__":
    app = QtWidgets.QApplication([])

    widget = TextWidget()
    widget.show()

    sys.exit(app.exec())
```

First, import the necessary libraries:

```
import sys
from PySide6 import QtCore, QtWidgets
```

Then, create a widget class that inherits from `QtWidgets.QWidget`:

```
class TextWidget(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
```

Inside the widget class, create one label and one button:

```
        self.label = QtWidgets.QLabel("Hello Pyside")
        self.button = QtWidgets.QPushButton("Button")
```

Then, arrange them inside the layout of the widget:

```
        self.layout = QtWidgets.QVBoxLayout(self)
        self.layout.addWidget(self.label)
        self.layout.addWidget(self.button)
```

The layout inside the widget must be named `self.layout`. In this example, you add the label and the button inside the vertical box layout, `QVBoxLayout`.

Then you can create a main script block:

```
if __name__ == "__main__":
```

Inside the block, initialize the GUI application:

```
    app = QtWidgets.QApplication([])
```

You initialize the widget class that you have defined and display it using the `show` method:

```
    widget = TextWidget()
    widget.show()
```

Finally, execute the GUI application:

```
    sys.exit(app.exec())
```

You can run the script this way:

```
(.venv) $ python hello_pyside.py
```

The following GUI application will show up.

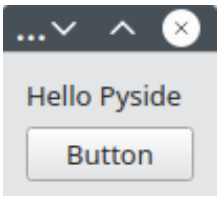


Figure 7.1: A GUI application using PySide

You may try clicking the button, but you'll quickly find out that it doesn't do anything. To make the button do something, you need to create a slot and connect the button to the slot. Let's create another script so that the button has a purpose. Name the script `hello_button.py` and add the following code to it:

```
import sys
from PySide6 import QtCore, QtWidgets


class ButtonWidget(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()

        self.button = QtWidgets.QPushButton("Button")

        self.layout = QtWidgets.QVBoxLayout(self)
        self.layout.addWidget(self.button)

        self.button.clicked.connect(self.button_on_clicked)

    @QtCore.Slot()
    def button_on_clicked(self):
        print("Clicking the button")
```

```
if __name__ == "__main__":
    app = QtWidgets.QApplication([])

    widget = ButtonWidget()
    widget.show()

    sys.exit(app.exec())
```

You can create a slot with the `@QtCore.Slot` annotation and annotate the function with it:

```
    @QtCore.Slot()
    def button_on_clicked(self):
        print("Clicking the button")
```

Then, you connect the button with this slot in this way:

```
        self.button.clicked.connect(self.button_on_clicked)
```

You get the `clicked` signal instance from the `button` object and you can connect it with the function that you've annotated with the slot.

You can launch the GUI application with a button in this way:

```
(.venv) $ python hello_button.py
```

Then, you will get this GUI application:

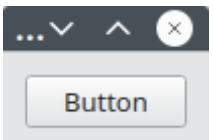


Figure 7.2: A GUI application with a button

Now, you can press the button. You will see this message printed:

```
Clicking the button
```

But what if you want to change the widget's property from the button's callback? For example, you want to change the label by clicking the button.

Let's create another script to demonstrate this purpose. Name the script `simple_app.py` and add the following code to it:

```
import sys
from PySide6 import QtCore, QtWidgets
```

Here, you import the necessary libraries.

Next, you should create a widget that combines a label, a text field, and a button:

```
class SimpleAppWidget(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()

        self.label = QtWidgets.QLabel("Hello")
        self.textedit = QtWidgets.QTextEdit("")
        self.button = QtWidgets.QPushButton("Say Hello")

        self.layout = QtWidgets.QVBoxLayout(self)
        self.layout.addWidget(self.label)
        self.layout.addWidget(self.textedit)
        self.layout.addWidget(self.button)

        self.button.clicked.connect(self.button_on_clicked)

    @QtCore.Slot()
    def button_on_clicked(self):
        text = self.textedit.toPlainText()
        self.label.setText(f"Hello {text}")
```

To add a text field, you create `QTextEdit`. Then, you add to the layout along with the label and the button. You also connect the button with the callback with a slot.

To get the text value from this text field, you use the `toPlainText` method. Then, you can set the value of the label with the `setText` method.

Lastly, you need to add the `main` script block:

```
if __name__ == "__main__":
    app = QtWidgets.QApplication([])

    widget = SimpleAppWidget()
    widget.show()

    sys.exit(app.exec())
```

You can run the script in this way:

```
(.venv) $ python simple_app.py
```

You will get the following window, which shows a label, a text field, and a button:

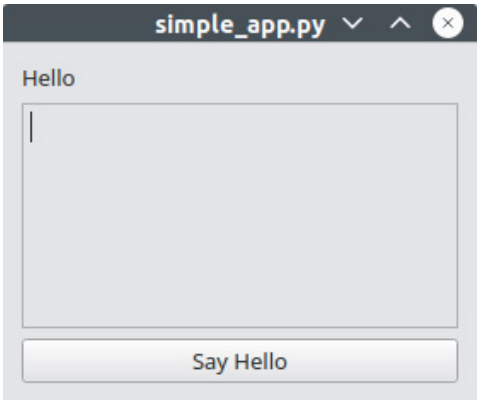


Figure 7.3: A GUI application with a button, a text field, and a label

Type `Ethereum` in the text field then press the button. The label will be changed just like the following screen:

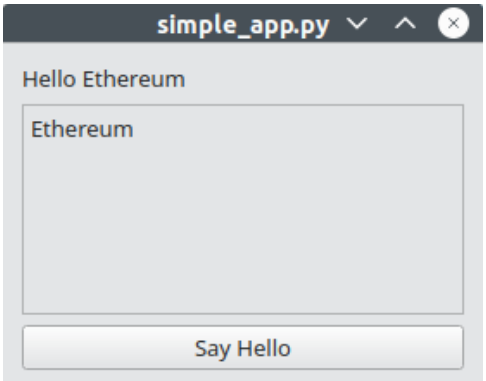


Figure 7.4: The changed label after clicking the button

Now that you understand the basics of creating a GUI application, it's time to create the front-end application of the smart contract.

Creating the voting GUI application

There are some aspects to the voting application, such as adding proposals, giving the right to vote, and voting action itself. But you're going to focus on the voting process itself. You're going to build the GUI application to vote for a proposal as a voter. To vote means to choose a proposal. So, you need a widget to choose a proposal. The best widget to do that is a combo box.

So, let's create a script that has a combo box and a button. In addition to that, let's add a label to the GUI application so your application has a title. This time, you want the script put inside the `scripts` folder inside your Ape project so that later you can integrate it with smart contract libraries. Name the script `voting_gui_app.py` and add the following code to it:

```
from ape import accounts, project
import os, sys
from PySide6 import QtWidgets


class VotingWidget(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()

        self.label = QtWidgets.QLabel("Voting Blockchain App")

        self.combobox = QtWidgets.QComboBox()
        self.combobox.addItems(["---", "beach", "mountain"])

        self.button = QtWidgets.QPushButton("Vote")

        self.layout = QtWidgets.QVBoxLayout(self)
        self.layout.addWidget(self.label)
        self.layout.addWidget(self.combobox)
        self.layout.addWidget(self.button)


def main():
    app = QtWidgets.QApplication([])

    widget = VotingWidget()
    widget.show()

    sys.exit(app.exec())
```

As you can see, you launch the GUI application with the Ape script using the `main` function. Inside the widget, you create a combo box using `QComboBox`. To add options to this combo box, you use the `addItems` method, which accepts a list of values for the combo box.

To run the script, you can run it this way:

```
(.venv) $ ape run voting_gui_app
```

You'll get this window:

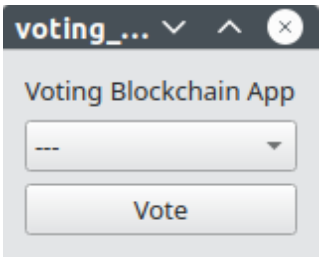


Figure 7.5: The voting blockchain application window

You can click the combo box and see some options.

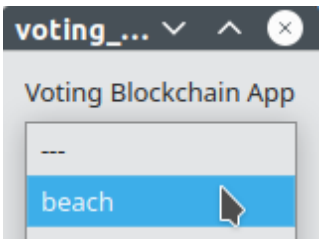


Figure 7.6: The combo box

But nothing happens if you click the **Vote** button. So, you need to add the callback to the button. You also need to get the selected value from the combo box.

You need to create a method inside `VotingWidget` as the callback for the button. Put this code inside `VotingWidget` and below the `__init__` method:

```
    @QtCore.Slot()
    def button_on_clicked(self):
        proposal = self.combobox.currentText()
        index = self.combobox.currentIndex()
        print(f"The selected proposal is {proposal}")
        print(f"The selected index is {index}")
```

Inside the callback, you get the selected text and the selected index from the combo box.

To connect the button with this callback, add this line of code to the bottom of the `__init__` method:

```
        self.button.clicked.connect(self.button_on_clicked)
```

Then, run the script again:

```
(.venv) $ ape run voting_gui_app
```

If you choose `mountain` in the combo box and then press the **Vote** button, you will get this message printed on the terminal:

```
The selected proposal is mountain

The selected index is 2
```

Now that you've created the GUI application, it's time to connect the GUI application with the smart contract.

Connecting the GUI application with the smart contract

Since the script is already an `ape` script, you can connect to the smart contract directly. There are three smart contract functions that this voting GUI application needs. They are the functions to get the number of proposals, read the proposal names, and vote for a proposal.

Right now, you hardcoded the proposals inside the combo box. But you want to get them from the smart contract.

Edit `voting_gui_app.py`; locate this line:

```
self.combobox.addItems(["---", "beach", "mountain"])
```

Change it to this line:

```
self.combobox.addItems(["---"])
```

You don't want to hardcode the options on the combo box anymore.

Then, you replace all the code inside the `main` function with the following code:

```
def main():
    voting_app_address = os.environ["VOTING_APP_ADDRESS"]
    contract = project.VotingApp.at(voting_app_address)
    amount_proposals = contract.amountProposals()

    app = QtWidgets.QApplication([])

    widget = VotingWidget()
    widget.show()

    for i in range(amount_proposals):
```

```
        proposal_name = contract.proposals(i).name
        widget.combobox.addItem(proposal_name)

    sys.exit(app.exec())
```

You can initialize the contract from the address defined in the environment variable. Then, you get the total amount of the proposals inside the voting smart contract with the `amountProposals` method. To get the proposal's name, you get it from the `name` property from the `proposals` hashmap. Then, you use the `addItem` method from the combo box to add the proposal's name to the combo box.

This time, you must define which network your voting blockchain app is running when running the script:

```
(.venv) $ ape run voting_gui_app --network ethereum:local:geth
```

You'll get the same GUI application but this time the options come from the smart contract.

Now, let's connect the `vote` method from the smart contract to the GUI application. Edit the `voting_gui_app.py` file and add the following code to the bottom of the `button_on_clicked` definition:

```
        password = os.environ["VOTER_PASSWORD"]
        address = os.environ["VOTING_APP_ADDRESS"]
        voter_account = os.environ["VOTER_ACCOUNT"]
        voter = accounts.load(voter_account)
        voter.set_autosign(True, passphrase=password)

        contract = project.VotingApp.at(address)
        contract.vote(index-1, sender=voter)
```

You can set up the Ethereum account to be used to vote. To do that, you need the account name and the password. After that, you call the `vote` method with the argument of the proposal index. In this case, the index must be subtracted by 1 because you put a dummy option on the combo box.

Launch the application again:

```
(.venv) $ ape run voting_gui_app --network ethereum:local:geth
```

Choose `mountain` and then press the `Vote` button and you will get this output:

```
INFO: Connecting to existing Geth node at http://127.0.0.1:8545/
[hidden].
The selected proposal is mountain
The selected index is 2
WARNING: Danger! This account will now sign any transaction it's
given.
WARNING: Using cached key for voter1
INFO: Confirmed
0x1be066788cad252b2f4d6f4e3f1d475b723daad0b302ac638a52f57b331dcf0a
(total fees paid = 21214060379384)
```

As you can see, you have successfully voted in the smart contract through the GUI application!

This GUI application is not perfect. It's better if you get the account using the login form instead of taking it from environment variables. You also need to add features for the chairperson to add proposals and add the right to vote.

However, this GUI application can be a starter for your next journey in building a decentralized frontend application.

Summary

In this chapter, we started by writing scripts to interact with the voting smart contract developed in the previous chapter. We noticed that interacting with smart contracts by using scripts is clumsy and not user friendly. Then, we installed the `PySide` library in order to write a decentralized frontend application for our voting smart contract. We learned about the library by building small GUI applications and understood the way to compose widgets and callbacks. After that, we built the GUI application for choosing proposals from the smart contract. But there is so much to building frontend applications that this chapter is not enough to explain all of them. So, let's build another frontend application. In the next chapter, we're going to build a simple Ethereum wallet to learn more about building frontend applications.

8
Cryptocurrency Wallet

In this chapter, we will learn how to develop a cryptocurrency wallet using the `PySide` library that we've used in the previous chapter. Although this is another front-end decentralized project, we'll learn new knowledge when building this project. So, we'll not be repeating what we've done previously.

Cryptocurrency wallet is a popular application for crypto users. It's a crucial one because this is where users hold their cryptocurrencies.

The following topics will be covered in this chapter:

- What is a wallet?
- Developing a cryptocurrency wallet

Technical requirements

The code for this chapter can be found in `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_8`.

What is a wallet?

A wallet is an application to hold and send cryptocurrencies. That's the basic tenet. It's similar to a real-life wallet that you put inside your pocket. You hold some money inside the wallet. The wallet keeps the money safe. You can also take out the money from the wallet to pay for some meals or other nice things.

Cryptocurrencies, such as Bitcoin or Ethereum are do not exist in the physical world. So, *what do we mean by holding cryptocurrencies in a cryptocurrency wallet?* It means you know the private key directly or indirectly (through the encrypted file with password). That way, you can create and sign transactions, like sending ETH or vote in a voting application. Thus, in one way or another, the wallet must provide access to the private key. Besides that, the wallet must show the balance of the account to its owner.

While cryptocurrency is digital and does not have a physical shape, the cryptocurrency wallet can have a physical shape. It's called a **hardware wallet**. Some of the examples are Ledger and Trezor wallets. It's a small hardware where you can generate the private key and check the balance.

But most of the cryptocurrency wallets exist in the form of software. The most famous example is MetaMask, which is a **browser extension wallet**. You must use browsers such as Firefox or Google Chrome in order to use this kind of wallet. With the browser extension, you can even interact with web applications that interact with blockchain. You can turn the voting application that you've developed in the previous chapter into a blockchain voting web application. Instead of using scripts or the GUI application, you interact with the web application to give a vote to a proposal.

It is possible for a cryptocurrency wallet to be a mobile application. MetaMask is also available as a mobile application. You can generate a private key and send cryptocurrencies through the mobile wallet application.

A cryptocurrency wallet can also be a desktop application. In this chapter, you're going to build a wallet as a desktop application. You still use the PySide library to build the front-end side. The wallet desktop application should be able to generate private keys and send cryptocurrencies.

In short, let's explore what a cryptocurrency wallet should be able to do?

- Generate accounts or private keys
- Show balance
- Send cryptocurrencies
- Create transactions on smart contracts

Other than core functions, it would be nice if the cryptocurrency wallet had some optional functions such as:

- Having an address book
- Showing the transactions history
- Updating the amount of the amount of gas and gas fee of a transaction

Of course, a wallet is a type of software that requires paramount security. Because if the managed private keys are leaking, then all your cryptocurrencies can be drained. Ideally, the private key should not be kept at all, reducing the risk. But this is not practical. You cannot expect a user to paste their private key every time they make a transaction. The process of copying the private key from somewhere and pasting to the cryptocurrency wallet is prone to leaking. So, for practical purposes, you have to keep the private key in your wallet application. But saving the private key in plain text is a recipe for disaster. Instead, you should save it in encrypted format.

You've seen this when you created accounts with the Ape Framework. The result of the created account is an encrypted file. The content looks like this:

```
{"address": "d8df097c9b1c901fa8ff32b8cbcf641fe5e35055",
"crypto": {"cipher": "aes-128-ctr", "cipherparams":
{"iv": "ed02728cd1be49dee4d982c79e104cb0"}, "ciphertext":
"4cb99721e91333cb3392f33901081848fafbab9bc4c2688936cc339633c5e038",
"kdf": "scrypt", "kdfparams": {"dklen": 32, "n": 262144, "r":
1, "p": 8, "salt": "f40f8240d6f24a6b4c9f7decea27009b"}, "mac":
"df57154c6e5af79edd848fb8086d54c274ba2f138c648b38e7308ac47ba3f1ac"},
"id": "d85a4817-2e9e-46ec-a8d6-f41f45746c64", "version": 3}
```

This encrypted file corresponds to the address `0xd8df097c9b1c901fa8ff32b8cbcf641fe5e35055` as seen in the address field. However, the private key is hidden inside the crypto field. To unlock this encrypted value, you must know the password to this encrypted value. When you generate the Ethereum account with the Ape Framework, you should've been asked what password you want to use.

Most of cryptocurrency wallets will use this approach. When you generate the account or the private key, they will be kept inside the application but encrypted. You only need to remember the password. When you open the cryptocurrency wallet, you'll be asked the password not the private key. The password you use will unlock the encrypted private key file and load the private key to memory for temporary use so you can make transactions. But if the cryptocurrency wallet is idle for a while, the cryptocurrency wallet will be locked and you will be asked your password again to unlock it again.

Other than that, the cryptocurrency should be able to export the private key for the backup purpose. Also, you should be able to import your cryptocurrency account with your private key to the cryptocurrency wallet.

Another important function of a cryptocurrency wallet is to generate an account, hence generating a private key. How do you generate a private key? A private key is just a number between 1 and 2 to the power of 256 minus one. So, you could just generate a random number in this range.

An example of a private key in hexadecimal is `7bd07c5ceb001faec7978e9972cceaefc52ed4a01068db8b697fc0e4db807b7c`. The length of this key is 64 characters. So, you can iterate 64 times to generate a random hexadecimal number between 0 and f.

Generating a private key looks simple. But in order to create a truly secure private key, you must have a secure source of randomness. The private key that a cryptocurrency wallet generates must be random enough that other people cannot guess. If people can guess the private key, then they can know the private key and use the account.

Mnemonic keys

If you've used a cryptocurrency wallet and generated an account, most likely the wallet will not show you the private key, but the **mnemonic keys**. It will force you to confirm that you have backed up the mnemonic keys before it lets you use the wallet.

You can say the mnemonic keys are similar to the private key. But the difference is the mnemonic keys are easier to remember.

> **Note**
>
> An example of mnemonic keys is "apple banana candle diamond elephant feather galaxy honey iceberg jigsaw kite lemon mango necklace octopus penguin queen rainbow sunflower tiger umbrella volcano whale zebra". This is 24-word mnemonic keys. Each word comes from a dictionary that has 2048 words. So there is a combination of 2048 to the power of 24 or 296 427748447529460284341721622241044104371160744039843941011415060257611878236 16. This is a very large number.

However, the mnemonic keys are not a private key. They are not another form of a private key. Instead, from mnemonic keys you can derive private keys as many as you need. But you cannot derive the mnemonic keys from a private key. So a private key corresponds to a cryptocurrency account but a series of mnemonic keys can correspond to many cryptocurrency accounts. The cryptocurrency wallet should have a way to create other accounts easily.

Now that you have a basic understanding of what a cryptocurrency wallet is, let's develop a simple cryptocurrency wallet. This application will not be a full-blown wallet application. You will develop a cryptocurrency wallet as a desktop application.

Developing a cryptocurrency wallet

To make things easier for you, you should base your wallet application with the Ape Framework that you've learned in the previous chapters. Your wallet application will interact with the blockchain. Then you already know how to interact with the blockchain using the Ape Framework.

In the previous section, you learned that one of many responsibilities of a cryptocurrency wallet is to generate cryptocurrency account or implicitly generate mnemonic keys or private keys.

To simplify the scope of the project, you will not develop that feature. You'll let the Ape Framework handle that case. But of course, after finishing this chapter, you're free to incorporate that feature into your cryptocurrency wallet. Instead, the features that you will develop are transferring ETH, saving target addresses to a local database, and checking and updating balance automatically.

For starters, let's create a simple application to transfer ETH to another address. That's the only purpose of the application.

Set up the Python environment and install the necessary libraries:

```
$ python3.10 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install --upgrade pip
(.venv) $ pip install eth-ape'[recommended-plugins]'==0.7.23
```

Create an Ape project:

```
(.venv) $ mkdir wallet
(.venv) $ cd wallet
(.venv) $ ape init
Please enter project name: Wallet
SUCCESS: Wallet is written in ape-config.yaml
```

Create a new file inside the scripts folder and name it `wallet_transfer.py`. First, add the libraries:

```
import sys, os
from PySide6 import QtCore, QtWidgets
from ape import accounts
```

Then you create a widget class:

```
class TransferWidget(QtWidgets.QWidget):
    username = os.environ["WALLET_USERNAME"]
    password = os.environ["WALLET_PASSWORD"]
```

In the widget class, you get the wallet username and password from environment variables and store them into the state variables. The username refers to the encrypted private key file generated by the Ape Framework.

You can generate an Ethereum account with the Ape Framework this way:

```
(.venv) $ ape accounts generate user1
```

The result can be found in the `~/.ape/accounts` folder:

```
$ ls ~/.ape/accounts/user1.json
```

When generating the account, of course you'll be prompted a password.

Then store the username, user1, to the `WALLET_USERNAME` environment variable and the password to the `WALLET_PASSWORD` environment variable.

```
(.venv) $ export WALLET_USERNAME=user1
(.venv) $ export WALLET_PASSWORD=yourpassword
```

Then you should create the constructor method where you generate the ETH amount field, the target address field, and the transfer button on the bottom of the `wallet_transfer.py` file by adding the following code:

```
    def __init__(self):
        super().__init__()
```

```
        eth_label = QtWidgets.QLabel("ETH")
        self.amount_field = QtWidgets.QTextEdit("")
        self.amount_field.setFixedHeight(30)
        amount_layout = QtWidgets.QHBoxLayout()
        amount_layout.addWidget(self.amount_field)
        amount_layout.addWidget(eth_label)

        self.address_field = QtWidgets.QTextEdit("")
        self.address_field.setFixedHeight(30)
        self.address_field.setPlaceholderText("Address")
        self.transfer_button = QtWidgets.QPushButton("Transfer")

        self.layout = QtWidgets.QVBoxLayout(self)
        self.layout.addLayout(amount_layout)
        self.layout.addWidget(self.address_field)
        self.layout.addWidget(self.transfer_button)

        self.transfer_button.clicked.connect(self.transfer_button_on_
clicked)
```

For the ETH amount field, you use `QHBoxLayout` to group the amount field and the ETH field. While `QVBoxLayout` is a vertical layout where you can stack widgets vertically, `QHBoxLayout` is a horizontal layout where you can stack widgets horizontally. But you can nest `QVBoxLayout` inside `QHBoxLayout` or vice versa. So the main layout, `self.layout`, is a vertical layout. Then you add the horizontal layout in this vertical layout using the `addLayout` method as can be seen in the code below:

```
        self.layout.addLayout(amount_layout)
```

This is different compared to adding a widget to a layout. You use the `addWidget` method for that purpose.

To create a text input field, you can use `QTextEdit`. You can alter the height with the `setFixedHeight` method and add a placeholder so a user can know what to fill in this text input field with the `setPlaceholderText` method.

Lastly, you connect the button to the callback function which you will define later.

The button to transfer needs a callback. Add the following code to define the callback:

```
    @QtCore.Slot()
    def transfer_button_on_clicked(self):
        address = self.address_field.toPlainText()
        amount = self.amount_field.toPlainText()
        amount = amount + "000000000000000000"
```

```
        wallet_account = accounts.load(self.username)
        wallet_account.set_autosign(True, passphrase=self.password)
        wallet_account.transfer(address, amount)
        self.close()
```

Inside the callback, you get the values from the address field and the amount field from the input text fields with the `toPlainText` method. Because the amount is in ETH, you need to convert it to `wei`. The easy way is just to append 18 times of "0" to the amount value.

Then you load the account using the username and the password you've stored in the state variables.

To transfer ETH, you can use the `transfer` method. The method accepts the target address as the first argument and the amount (in `wei`) as the second argument. Lastly, you close the widget.

Then you can define the main method so you can call the script using the Ape command:

```
def main():
    app = QtWidgets.QApplication([])

    widget = TransferWidget()
    widget.show()

    sys.exit(app.exec())
```

You create a `PySide` application then you display the `TransferWidget` widget class. Finally, you can execute the `PySide` application using the `exec` method.

The script is now complete. Before you run the script, let's set up the development blockchain with `geth`, as shown here.

```
$ geth --dev --http --http.api eth,web3,net
```

Then you can get inside the Geth console:

```
$ geth attach /tmp/geth.ipc
```

In this development blockchain, the first account has all the ETH. You need the first account to send some ETH to the `user1` account. You can send some ETH from the first account to the user1 account with the `sendTransaction` method.

```
> eth.sendTransaction({from: eth.accounts[0], to:
"0xd8df097c9b1c901fa8ff32b8cbcf641fe5e35055", value: web3.toWei(50,
"ether")})
```

Of course, you must change the `0xd8df097c9b1c901fa8ff32b8cbcf641fe5e35055` address to your **user1** address.

Now that your `user1` account has some ETH, run the `wallet_transfer.py` script:

```
(.venv) $ ape run wallet_transfer --network ethereum:local:geth
```

You'll be greeted with this window:

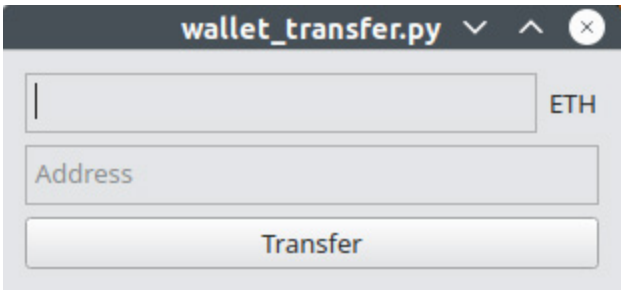


Figure 8.1: The transfer window

Your user1 account has 50 ETH. So you should fill any number not exceeding 50 in the amount field. You can put 5. Then put an Ethereum address in the address field. You can generate another Ethereum account to get a new address:

```
(.venv) $ ape accounts generate user2
```

The window should look like this:

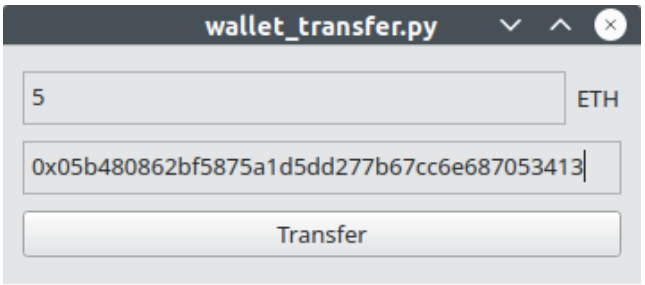


Figure 8.2: The transfer window with fields filled with values

Finally, you can click the `Transfer` button. The application will close.

You can check the balance of the target address in the Geth console to make sure the transfer works.

```
> web3.fromWei(eth.
getBalance("0x05b480862bf5875a1d5dd277b67cc6e687053413"))


5
```

So you have created a GUI application that can transfer ETH from one address to another address.

However, pasting the target address like this is prone to error. If you miss a character, money will be lost. So, it's important to devote attention to the user experience (UX) to make sure the cryptocurrency wallet application can be used in a more easy and convenient way.

Wallet UX

Inputting an ETH address every time you want to send ETH is bad user experience. The problem is the Ethereum address looks very random. It's like a bank account number but longer and more cryptic. However, to reduce the chance of sending money to wrong accounts, you need to provide the name of the owner of the bank account when filling the destination other than the bank account number. A name is tied to a bank account number. When you want to send some money to your grandma, you can mistake your grandma's bank account number 33445566 with 33445567, but not her name.

You can take this approach when developing a cryptocurrency wallet. You can tag a label to an Ethereum account address. That way, you don't have to paste an Ethereum address every time you want to transfer ETH. This is very useful if you're going to transfer ETH very often to some addresses.

So let's build a simple application that can add addresses to a simple database file. Create a file inside the `scripts` folder and name `it add_wallet_to_db.py`. Then add the following import statements:

```
import sys, os, json
from PySide6 import QtCore, QtWidgets
from ape import accounts
```

For this simple application, you're just going to save addresses and their labels to a simple `json` file. So let's create a widget class and declare the state variables pointing to the location of the `json` file by adding the following code:

```
class AddAddressToDbWidget(QtWidgets.QWidget):

    home_dir = os.path.expanduser('~')
    wallet_db_path = f"{home_dir}/.wallet.json"
```

Then you need to define the widgets and the layout. Usually, you use the combination of `QHBoxLayout` and `QVBoxLayout` but there is an alternative. Add the following content to the file:

```
    def __init__(self):
        super().__init__()

        self.layout = QtWidgets.QFormLayout(self)
        self.save_button = QtWidgets.QPushButton("Save")
        self.address_field = QtWidgets.QTextEdit("")
        self.address_field.setFixedHeight(30)
```

```
        self.label_field = QtWidgets.QTextEdit("")
        self.label_field.setFixedHeight(30)
        self.layout.addRow("Address:", self.address_field)
        self.layout.addRow("Label:", self.label_field)
        self.layout.addRow("", self.save_button)

        self.save_button.clicked.connect(self.save_button_on_clicked)
```

In the code above, you use `QFormLayout` to arrange the widgets. While you stack widgets in a list with `QVBoxLayout` or `QHBoxLayout`, with the `QFormLayout` layout, you add one row at a time. In this example, each row consists of two columns. But it can be another thing like a layout. The first column is a label and the second column is a field. In this code, the fields are the text input and the button.

To add a row, you use the `addRow` method. You can give a string as a label for the first argument. Then you can give a widget for the second argument. Lastly, in this constructor method, you connect the button with a callback method.

Then you need to define the callback method by adding the following code to the file:

```
    @QtCore.Slot()
    def save_button_on_clicked(self):
        address = self.address_field.toPlainText()
        label = self.label_field.toPlainText()
        if not os.path.exists(self.wallet_db_path):
            data = {}
        else:
            with open(self.wallet_db_path, "r") as f:
                data = json.load(f)
        data[label] = address
        with open(self.wallet_db_path, "w") as f:
            json.dump(data, f)
        self.close()
```

You will get the label value and the address value from the address and label fields. Then you can load the database from the `json` file before adding the new label and address to the database. Lastly, you can write the database back to the `json` file before closing the application.

Then you need to create the main method to launch the widget class by adding the following code to the file:

```
def main():
    app = QtWidgets.QApplication([])

    widget = AddAddressToDbWidget()
```

```
    widget.show()

    sys.exit(app.exec())
```

Then you can launch the script this way:

```
(.venv) $ ape run add_wallet_to_db
```

You would get this window showing up:

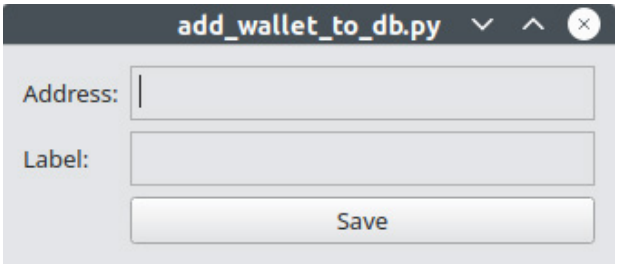


Figure 8.3: The window to add an address and its label

Add an address and a label before clicking the **Save** button as shown in the following image:

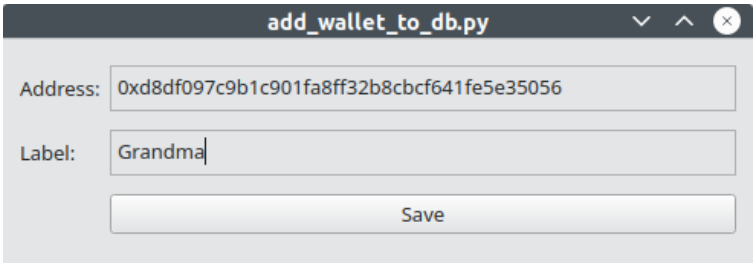


Figure 8.4: The window with filled fields

After clicking the **Save** button, the window will close. You can check whether it has saved your grandma's address by displaying the content of the `json` file:

```
$ cat ~/.wallet.json
{"Grandma": "0xd8df097c9b1c901fa8ff32b8cbcf641fe5e35056"}
```

Now that you have a local addresses database, let's revamp your transferring ETH application with this database so you don't have to fill the raw address again.

Let's create another script inside the scripts folder and name it wallet_transfer_address_from_db.py. Add the following imports to the file:

```
import sys, os, json
from PySide6 import QtCore, QtWidgets
from ape import accounts
```

Then you need to create a widget class for transferring ETH with addresses from the database file:

```
class TransferWidget(QtWidgets.QWidget):
    username = os.environ["WALLET_USERNAME"]
    password = os.environ["WALLET_PASSWORD"]
    home_dir = os.path.expanduser('~')
    wallet_db_path = f"{home_dir}/.wallet.json"
```

In this widget class, you load the Ethereum account's username and password from the environment variables and store them to state variables. Then you define the path of the json database file.

You need to define the widgets inside the layout of this widget class. First, you define the amount field:

```
    def __init__(self):
        super().__init__()

        eth_label = QtWidgets.QLabel("ETH")
        self.amount_field = QtWidgets.QTextEdit("")
        self.amount_field.setFixedHeight(30)
        amount_layout = QtWidgets.QHBoxLayout()
        amount_layout.addWidget(self.amount_field)
        amount_layout.addWidget(eth_label)
```

Then you create a combo box and load up the option values from the JSON database file. Add the code inside the `__init__` method on the bottom:

```
        self.address_combobox = QtWidgets.QComboBox()
        if not os.path.exists(self.wallet_db_path):
            data = {}
        else:
            with open(self.wallet_db_path, "r") as f:
                data = json.load(f)
        options = ["---"]
        for k,v in data.items():
            options.append(k + " - " + v)
        self.address_combobox.addItems(options)
```

You define the combo box with the `QComboBox` method. To add options to the combo box, you can use the `addItems` method with the array of strings as the argument. To get the values, you take them from the database `json` file with the `json.load` method.

Then you need to define the transfer button and add the callback to it. Add the code on the bottom of the `__init__` method:

```
        self.transfer_button = QtWidgets.QPushButton("Transfer")
        self.transfer_button.clicked.connect(self.transfer_button_on_
clicked)
```

You can the callback later.

Lastly, you define the vertical layout for the widget and add the amount field, the combo box and the button. Add the following code on the bottom of the `__init__` method:

```
        self.layout = QtWidgets.QVBoxLayout(self)
        self.layout.addLayout(amount_layout)
        self.layout.addWidget(self.address_combobox)
        self.layout.addWidget(self.transfer_button)
```

Then you must define the `callback` method of the transfer button:

```
    @QtCore.Slot()
    def transfer_button_on_clicked(self):
        label_address = self.address_combobox.currentText()
        if label_address == "---":
            return
        label, address = label_address.split(" - ")
        amount = self.amount_field.toPlainText()
        amount = amount + "000000000000000000"
        wallet_account = accounts.load(self.username)
        wallet_account.set_autosign(True, passphrase=self.password)
        wallet_account.transfer(address, amount)
        self.close()
```

You will get the address from the combo box and the amount from the amount field. You can prepare the account with the username and account. Then you call the `transfer` method with the destination address as the first argument and the amount as the second argument.

Lastly, you need to define the main method so you can call the script with the `ape` command. Add the following code at the bottom of the script file:

```
def main():
    app = QtWidgets.QApplication([])
```

```
    widget = TransferWidget()
    widget.show()

    sys.exit(app.exec())
```

Before running the script, let's clean the state of the development blockchain again. Stop the Geth blockchain development and run it again:

```
$ geth --dev --http --http.api eth,web3,net
```

Then you connect to the blockchain and bootstrap the balance of the sender:

```
$ geth attach /tmp/geth.ipc
> eth.sendTransaction({from: eth.accounts[0], to:
"0xd8df097c9b1c901fa8ff32b8cbcf641fe5e35055", value: web3.toWei(50,
"ether")})
```

Then you can run the script this way:

```
(.venv) $ export WALLET_USERNAME=user1
(.venv) $ export WALLET_PASSWORD=yourpassword
(.venv) $ ape run wallet_transfer_address_from_db --network
ethereum:local:geth
```

You would see a window that looks like this:

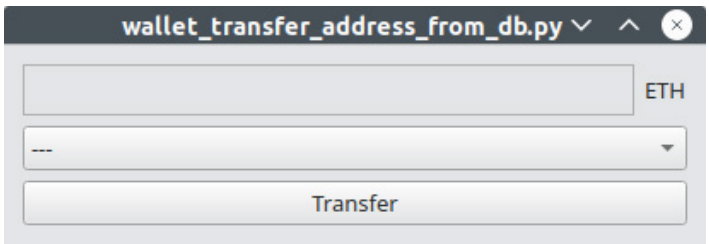


Figure 8.5: The transfer window with the address combo box

Select Grandma's address in the combo box and fill 7 in the amount field so the window looks like this:

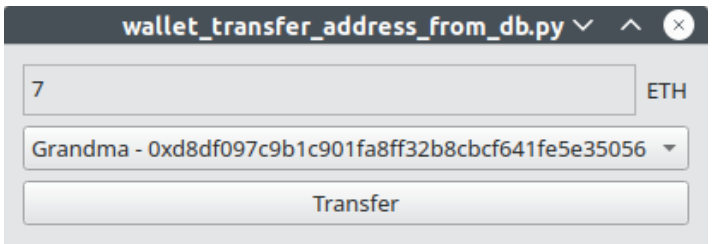


Figure 8.6: The window with selected and filled fields

Click the **Transfer** button. The window will close. To make sure the transfer takes place, you can check the balance of the sender and Grandma in the Geth console:

```
> web3.fromWei(eth.
getBalance("0xd8df097c9b1c901fa8ff32b8cbcf641fe5e35055"))
42.999983913494572
> web3.fromWei(eth.
getBalance("0xd8df097c9b1c901fa8ff32b8cbcf641fe5e35056"))
7
```

As you can see, the sender's balance decreased by 7 ETH (and a bit more because of the transaction fee) and the balance of Grandma's address increased by 7 ETH.

Sometimes when you want to transfer ETH to another address, you may want to check your balance to make sure you have enough ETH to transfer. Let's add the balance field that changes when there is an update to the balance.

Let's create another script named `full_wallet.py` inside the `scripts` folder. Add the full content of `wallet_transfer_address_from_db.py` to `full_wallet.py`.

Then you want to add the balance field to the existing script. First, add another import. Change this line:

```
import sys, os, json
```

To this line:

```
import sys, os, json, time
```

You're going to need the **time** library to delay checking the account balance from the blockchain. Using this library, you'll use the pooling technique to check the account balance. You cannot do it in the main thread because the main thread is busy updating User Interface (UI) so you need another thread to check the account balance. To do it, you'll create a thread class by adding the following code below the imports line to the script file:

```
class WorkerThread(QtCore.QThread):
    username = os.environ["WALLET_USERNAME"]
    update_label = QtCore.Signal(str)

    def run(self):
        while True:
            time.sleep(1)
            wallet_account = accounts.load(self.username)
            self.update_label.emit(str(wallet_account.balance))
```

You can use `QtCore.QThread` as the base class for this thread class, `WorkerThread`. A thread class is a blueprint of a thread on which you can define what you want to do when you create a thread. Then you have two state variables, the account username that you fetch from the environment variable and the signal so you can update the UI from this thread. The signal is a string type.

Then in the `run` method (the method that the thread will execute when it is launched), you loop forever with 1 second delay. You get the account balance using the **load** method from the **accounts** object. Then you send the signal with the `emit` method from the `QtCore.Signal` class with the account balance as the argument.

The next step is to add the balance field to the window. The balance field is just labels showing how much ETH this account has. Add the following code inside the __init__ method and below `the super().__init__() line`:

```
        balance_layout = QtWidgets.QHBoxLayout()
        balance_label = QtWidgets.QLabel("My Balance:")
        self.balance_field = QtWidgets.QLabel("ETH")
        balance_layout.addWidget(balance_label)
        balance_layout.addWidget(self.balance_field)
```

The balance widget is just a horizontal layout that is composed of one label showing the actual balance and the `"My Balance"` label.

Then you need to add this balance widget to the main layout. Add this line below the `self.layout = QtWidgets.QVBoxLayout(self)` line:

```
         self.layout.addLayout(balance_layout)
```

Then on the bottom of the `__init__` method, add the code to initialize the thread class and start it:

```
        self.thread = WorkerThread()
        self.thread.update_label.connect(self.update_balance)
        self.thread.start()
```

You initialize the thread class, `WorkerThread`, then run the thread with the `start` method. But before starting the thread, you need to connect the `Signal` object inside the thread class with the `connect` method to a function named `update_balance` of this widget class that you'll define later.

Then let's define the `update_balance` method inside the `TransferWidget` class:

```
    def update_balance(self, balance):
        balance = int(balance) / 10**18
        self.balance_field.setText(str(balance) + " ETH")
```

Inside the method, you change the `balance_field` label to the actual balance in ETH. So that's why you convert the balance argument to integer and divide it by `1000000000000000000`.

Then replace the `self.close()` line in the `transfer_button_on_clicked` method with the following code:

```
        self.address_combobox.setCurrentIndex(0)
        self.amount_field.setText("")
```

The script is finished. You can run it this way:

```
(.venv) $ ape run full_wallet --network ethereum:local:geth
```

You will see this window:

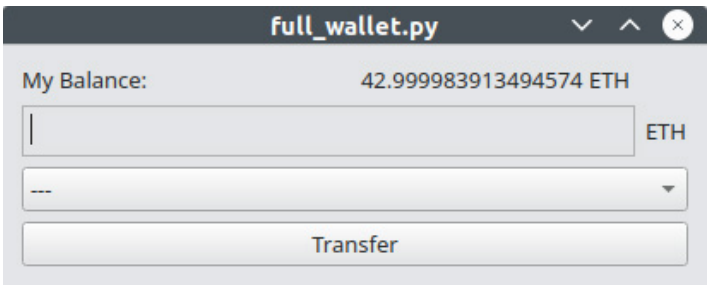


Figure 8.7: The window with shown balance

If you still use the same blockchain development, your balance will be close to 43 ETH because you just sent 7 ETH to Grandma.

Now select Grandma again from the address combo box and put 2 in the amount field so the window looks like this:

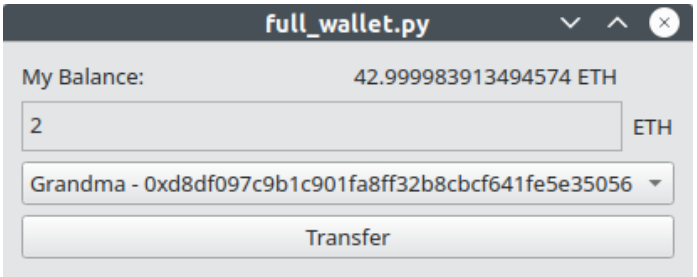


Figure 8.8: The window with filled values

Click the **Transfer** button and the balance will be updated automatically:

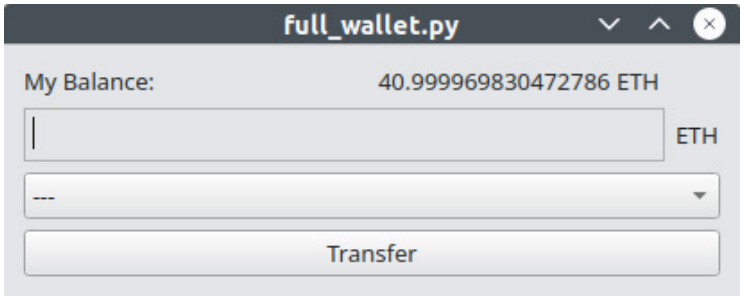


Figure 8.9: The window with the updated balance

As you can see the application to transfer ETH becomes easier to use with some improvements to the UX. But don't stop there. There are many improvements you can add to make the application much easier to use. You can display historical transactions of the account and the destination account. Then you should have the capability to change the gas and gas fee of the transaction. This application uses the default setting. A popular cryptocurrency wallet will give you this option in case you want to send the transaction as fast as possible with higher fee.

Of course, you should incorporate the creation of accounts and loading up the account using the password into your application. Right now, you use the environment variables which is not friendly for normal users. So there should be a login window before you can transfer ETH.

Also, in this chapter, you've developed a wallet that can send and transfer ETH, but not ERC20 tokens. For those who've never heard ERC20 tokens, you can heed to *Chapter 12* where we will discuss about ERC20 tokens. To add the sending ERC20 tokens feature to the wallet, it does not take additional much works. Sending ERC20 tokens is just executing functions on a smart contract like executing the voting function in the voting smart contract in the previous chapter. You can put the execution inside the button callback in the wallet.

Aside from improving UX, you can also make cryptocurrency wallet to be more secure. You've learned about encrypting the private key in the earlier section. But that's not the only way to secure the wallet. You can split the private key into many pieces with Secret Sharing algorithms. Also, you can create a cryptocurrency wallet in the form of the smart contract so to unlock the transaction you need M of N signatures. This is crucial if you are a clumsy person. If you forget your private key, there is a recovery procedure. You can ask other people who are responsible for other signatures to recover your account. If you want to be fancy, you can also add AI to digest the transactions that you want to sign. Is it dangerous or not? Is it malicious or not?

Summary

In this chapter, we learned what a cryptocurrency wallet is. We saw that a cryptocurrency wallet usually have certain features like generating accounts that protected by passwords, showing the balance of account, and so on. Then we developed a simple cryptocurrency wallet application using the PySide library and the Ape Framework. After testing it, we decided to add some improvements to the UX of the application like showing balance and using address database. Finally, we could feel the application become easier to use. In the next chapter, we will learn about InterPlanetary File System to overcome a certain weakness from blockchain.

Part 4: Related Technologies

This section delves into the fascinating world of related technologies that augment and intertwine with blockchain development. From the revolutionary InterPlanetary File System (IPFS) to scaling solutions such as Layer 2 protocols, each chapter explores how these technologies complement and enhance the decentralized landscape.

This section has the following chapters:

- *Chapter 9*, *InterPlanetary: A Brave New File System*
- *Chapter 10*, *Implementing a Decentralized Application Using IPFS*
- *Chapter 11*, *Exploring Layer 2*

9
InterPlanetary: A Brave New File System

In this chapter, we are going to learn about the **InterPlanetary File System** (**IPFS**). The IPFS is not actually part of the blockchain technology; however, it plays an important role in complementing blockchain. IPFS with blockchain is a match made in heaven. As you have learned in previous chapters, storage in a blockchain is expensive. Usually, people save links to files in a blockchain and save the actual files in normal storage, such as cloud storage. But this strategy suffers the fate of centralization. IPFS offers blockchain developers a way to avoid this.

In this chapter, you are going to learn about the following:

- The motivation behind IPFS
- Merkle DAG
- Peer-to-peer networking

Technical requirements

To follow this chapter, you need one of the more recent versions of Linux and Python — no older than 3.8. You could try on other platforms like Mac or Windows, but Linux is recommended. You could also run Linux with virtualization software like VirtualBox. We will use Ubuntu Linux 22.04 LTS and Python 3.10. The code is available at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_9`.

The motivation behind IPFS

IPFS is not a normal filesystem, such as `fat32` , `ntfs` , or `ext3` . It is more similar to Dropbox. It is a cross-device filesystem. You can save a file in this filesystem and people around the world can access it as easily as if the file were on their own computer. If Ethereum can be thought of as the world's singleton operating system, IPFS can be considered as the world's singleton storage!

The slogan of the IPFS website is IPFS is the Distributed Web. IPFS tries to replace, or at least supplement, HTTP. The HTTP protocol has served us for a long time, over 20 years, but it is not considered sufficient for upcoming challenges, such as increasing bandwidth demands or redundancy of files. HTTP uses a client-server model. You can only choose one of these two roles: either to be a server or a client.

There are a couple of problems with this architecture:

- The first problem is that to pick up the server role, we have to have sufficient resources. If not, the server is flooded with a lot of requests and it could go down rapidly. The resources required to handle one million requests per minute is out of reach for many common people.
- The second problem is that the server-and-client architecture is not efficient in some situations. Imagine that you are sitting beside your grand mother in a park and both of you are watching the same video of a cute panda from the same URL (something like `https://example.com/cute_ panda.mp4`). Let's say that the size of this video is 20 MB. This means the server must send a 20 MB file twice to two different locations, even though these two different locations are located closely together with a proximity of one meter. In other words, the server uses 40 MB of bandwidth. Imagine, however, if you could pull the file not from the server, but from your grand mother who sits beside you (in this case, let's assume grandma has watched the cute panda video two minutes before you). Wouldn't this be more efficient?

Juan Benet, who graduated with a Master's degree in Computer Science from Stanford University, was inspired to build IPFS in late 2013. Back then, he was working with knowledge tools, a term that refers to software that can be used to efficiently gather knowledge from papers. Let's say, for example, that a scientist reads a lot of papers. It would be better if that scientist could get this knowledge faster. Benet came across the problem that datasets required too much effort to distribute. There was no easy way to handle the versioning of datasets. He looked at various tools, such as Git and BitTorrent, and wondered if they could be combined to solve this problem. As a result, IPFS was born. BitTorrent inspired IPFS with regard to distributing files and finding files among the nodes. Git inspired IPFS with regard to keeping the integrity of files and converting saved files into storage.

IPFS is a peer-to-peer hypermedia protocol that makes the web faster, safer, and more open. The goal of IPFS is pragmatic and idealistic. Besides saving bandwidth, another of its aims is to increase the longevity of a file. Keeping a file in a server for a very long time (such as a decade) requires a huge amount of resources. The reason why we might want a file to stay alive is usually because it has some kind of economic benefit for the owner of the server; for example, it could be monetized with ads if it is a blog post. If not, there is a possibility that the file will be destroyed by the owner of the storage server. This happened when GeoCities was shut down.

Note

GeoCities was a website that allowed people to create their own personal website. It was similar to `wordpress.com` and `medium.com`. Some owners of servers would keep files alive even without ads, like Wikipedia, which works thanks to donations. Other than that, however, the files are not so lucky.

The other goals of IPFS are more idealistic and involve democratizing how we provide content. Right now, content is heavily centralized. We usually go to just a few websites, such as Facebook, Instagram, Reddit, Medium, Netflix, Amazon, Google, Wikipedia, and so on. This oligopoly of information hinders innovation on the internet because information is controlled literally by a few companies. Apart from Wikipedia, most, if not all, companies are beholden to rich shareholders. This situation is in stark contrast to 10 years ago, when the internet was considered a great equalizer of wealth and information, similar to printing press technology.

The other disadvantage of this heavy centralization is that the information that's provided is susceptible to censorship. For example, Google is a company based in Mountain View, California, and is therefore subject to US law. Most people who have the power to make decisions (senior executives and C-levels) tend to be American and therefore have an American bias in their perception of the world. Things that are fine in most countries in Europe could be censored in the name of American morals. This could include content that is disliked by the state because it is considered blasphemous or dangerous. The founder of the IPFS project likened this situation to the case of burning books that were considered dangerous by the state or powerful institutions. One of the goals of the IPFS project was to increase the resistance of documents to censorship. IPFS makes it easier for people to mirror and serve dangerous documents. We'll discuss how IPFS achieves this goal in a later section of this chapter.

The final goal of IPFS, which is more pragmatic, concerns our fragile internet infrastructure, which is composed of computer networks and core routers connected by fiber-optic cables. If the connecting cable is damaged accidentally or deliberately, a block or area could go offline. In 2011, a woman with a shovel damaged the cable that brought internet to Armenia when she was digging looking for metal to sell. The IPFS project does not solve this problem completely, but it can mitigate the damage to some extent.

Note

You can find the incident about the woman and her shovel here: `https://web.archive.org/web/20141225063937/http://www.wsj.com/articles/SB10001424052748704630004576249013084603344`.

The basic data structure in IPFS is Merkle DAG. Before you can understand the inner workings of IPFS, you need to learn about Merkle DAG.

Merkle DAG

If you have learned about the internals of Git**, Merkle Directed Acyclic Graph** (**DAG**) shouldn't be too foreign. As a version control system software, Git is required to keep many versions of a file and distribute them easily to other people. It also needs to be able to check the integrity of the file very quickly.

There are two words that make up Merkle DAG: Merkle and DAG. Let's discuss Merkle first. Actually, the full word of Merkle in this context is Merkle tree. A Merkle tree is a fast way to check whether partial data has been tampered with or not.

Merkle tree

Let's take a look at an example of a **Merkle tree** in order to understand it. Let's say you have eight pieces of data. In this case, we will use the names of animals for our data, but in Bitcoin, which uses a Merkle tree, the pieces of data are usually transactions. Back to Merkle trees: put the data in order, so in this case, cat is the first piece of data, dog is the second, ant is the third, and so on. See *Figure 9.1*:

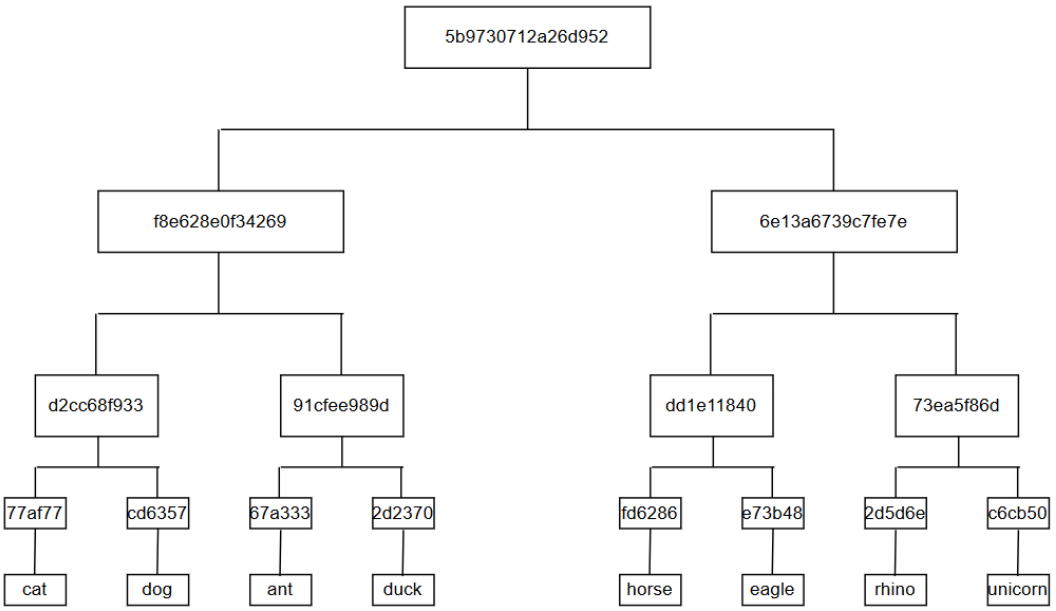


Figure 9.1: The tree hash

We take the hash of each piece of data in the tree hash in *Figure 9.1*, in this case, cat, dog, ant, and so on. For this demonstration, we use the hash function `SHA256`. Because of limited space, we have truncated the full hash result in the diagram. For now, we will order the data from left to right, so the hash of the "cat" string is Data 1, the hash of the "dog" string is Data 2, the hash of the "ant" string is Data 3, and so on.

Here's come the interesting part. For Data 1 and Data 2, we combine the hash and hash the result. Combining the hash means concatenating it. Do this for Data 3 and Data 4, Data 5 and Data 6, Data 7 and Data 8 as well.

This might remind you of a knockout competition. We are now in the semi-final phase. We now have Hash 1 (from Data 1 and Data 2), Hash 2 (from Data 3 and Data 4), Hash 3 (from Data 5 and Data 6), and Hash 4 (from Data 7 and Data 8).

We then concatenate Hash 1 and Hash 2, hash the result, and name this Hash 5. We then do the same thing for Hash 3 and Hash 4. Name the result Hash 6.

We are now in the final phase. Combine Hash 5 and Hash 6, then hash the result. The result is the Root Hash. This Root Hash can guarantee the integrity of all the pieces of data (from Data 1 to Data 8). If you change any of the data, Root Hash would be different.

You may be asking why we don't just concatenate all the data (from Data 1 to Data 8) from the beginning and then hash the result. It turns out, however, that Merkle trees has some benefits over just concatenating all the data together and then hashing it (this technique is called a hash list, and it is used in some situations). Using a Merkel tree offers the advantage of easier and more cost-effective verification of the integrity of partial data.

In a Merkle tree, to check the integrity of *Data 5*, you only need to download *Data 5*, *Data 6*, *Hash 4*, *Hash 5*, and the *Root Hash*, as shown in the following diagram. See Figure 9.2. You don't need to download all the data:

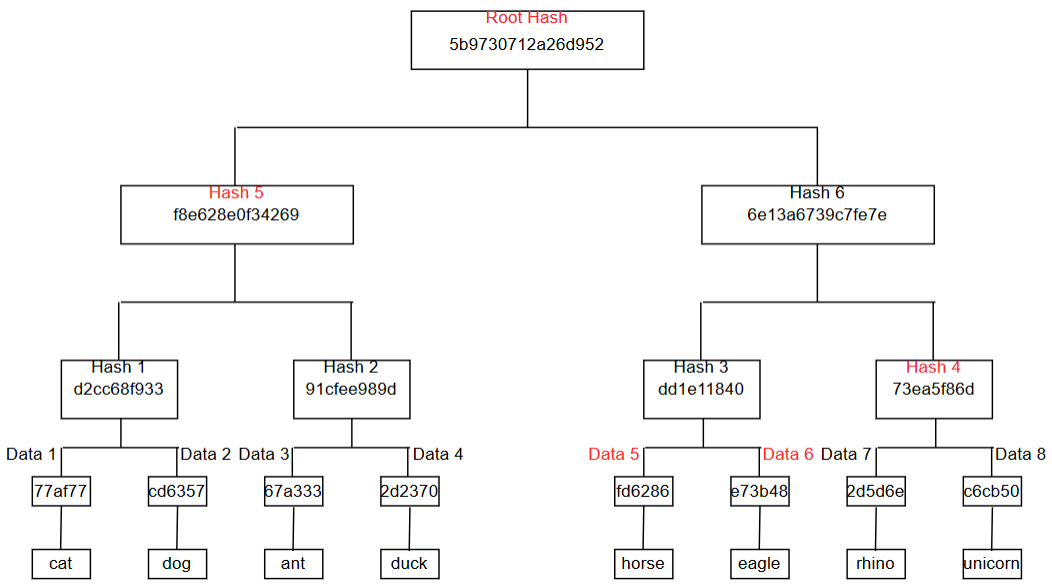


Figure 9.2: The Merkle tree

If you use a naive approach, you need to download all the hashes of the data (*Data 1* to *Data 8*) and the *Root Hash*. In this example, we only have eight pieces of data. Imagine if we had 100 and you had to download the entire dataset. Merkle tree makes this process more efficient because we don't need to download the full set of data.

If we had an odd number of nodes, such as seven, the general rule (the one that Bitcoin implements) is to clone the last node, so *Data 8* is a copy of *Data 7*. You could use another rule, however; I have seen an implementation of a Merkle tree in which a single piece of data (Data 7 in our example) is simply promoted to the top. In this case, Hash 4 is just Data 7.

This is what Bitcoin does when people use **Simplified Payment Verification**. With a mobile app, downloading the full node is difficult. In order to send Bitcoin transactions, the user downloads only the important parts of the node instead of the full node. Merkle tree enables this process.

In the next section, we will move on to learn about DAG.

Directed Acrylic Graph

Directed Acrylic Graphs (**DAG**), as its name suggests, are graphs in which each vertex (or node) can have edges pointing to other vertexes, as shown in the following diagram. See Figure 9.3:

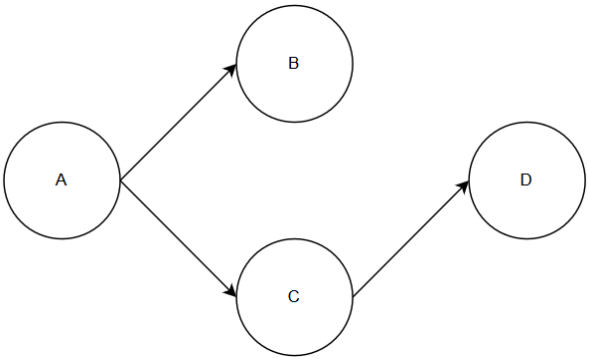


Figure 9.3: DAG with arrows pointing to the right

he direction of the arrow does not matter, as long as you make it consistent like in *Figure 9.4*:

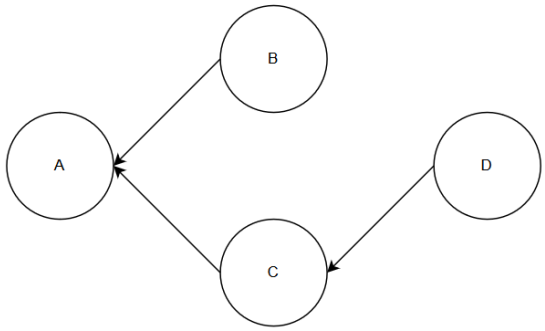


Figure 9.4: DAG with arrows pointing to the left

As shown in *Figure 9.3* and *Figure 9.4*, the rule is that these edges should not make a cycle. In the following figure, we can see that vertexes A, C, and D make a cycle, which is against the rules of a DAG. See *Figure 9.5*:

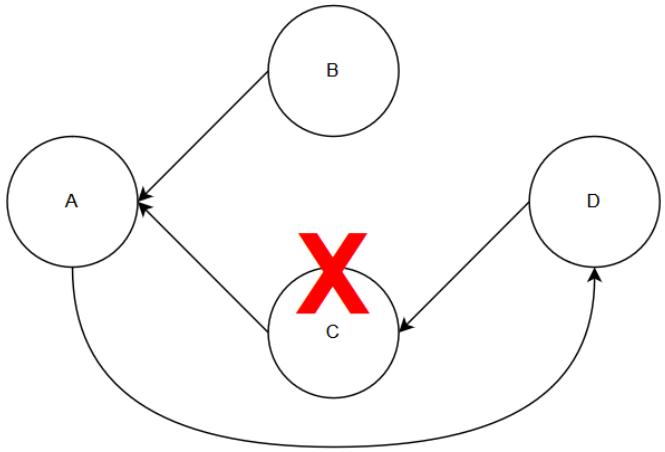


Figure 9.5: A cycle in a graph

Now, if you combine a Merkle tree and DAG, you get a **Merkle DAG**. This is the data structure that is used by Git and IPFS.

In a Merkle tree, only the leaf nodes hold data. In a Merkle DAG, however, any node could hold the data. In a Merkle tree, the tree has to be balanced, but there is no such limitation in a Merkle DAG.

Before we jump into Merkle DAGs, let's learn about content addressing, because Merkle DAGs are dependent on this feature.

Content addressing

In a linked list, you chain together nodes (or blocks) with a pointer. A pointer is a data type that points to memory. For example, let's say we have two nodes, node A and node B. Node A is the head and node B is the tail. The structure of the node has two important components. The first component is the data component where you store the data. In Git, this data could be the content of the file. The second component is a link to another node. In a linked list, this is the pointer to a node's address.

But with content addressing, instead of just a pointer, we also add the hash of the target (in this case, node B). You may recognize this concept; this is exactly what happens in blockchain. A Merkle DAG, however, is not a linked list that spans linearly in one straight line. A Merkle DAG is a tree that can have branches.

This is a linked list. It is used in a blockchain's data structure. See *Figure 9.6*:

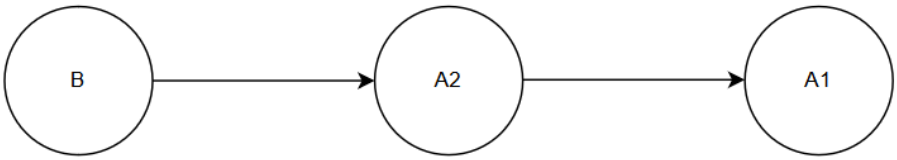


Figure 9.6: A linked list

Now, consider this case. We have three nodes: nodes A1 and A2 are both heads that point to node B. Instead of putting the pointers on node A1 and node A2, we put the pointers on node B. Node B now has two pointers. Node B hashes nodes A1 and A2, then concatenates both hashes before hashing the result again. In this way, node B can keep the integrity of the content of node A1 and node A2. If somebody changes the content of node A1 or the content of node A2, the hash kept by node B would be invalid. See *Figure 9.7*:

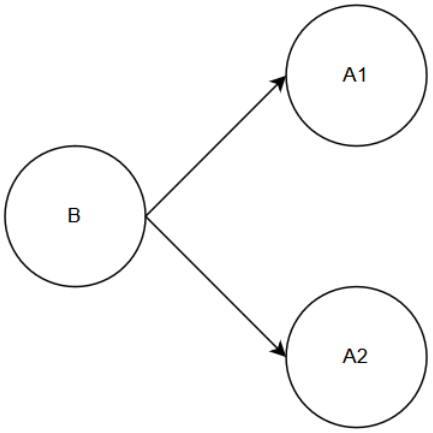


Figure 9.7: The node B pointing to the node A1 and A2

IPFS is different to HTTP in terms of how it fetches a document. HTTP uses links, which work like pointers. For example, let's say we have the following link: `https://example.com/cute_panda.png`. This uses a location to fetch a document called `cute_panda.png`. Only one provider could serve this document, which is `example.com`. IPFS, however, does not use a URL link. Instead, it uses a hash link, such as `ipfs://QmYeAiiK1UfB8MGLRefok1N7vBTyX8hGPuMXZ4Xq1DPyt7`. When you access this hash link, the IPFS software will find the document that, when hashed, will give you the same hash output. Because hashing is a one-way function, IPFS must have some other information to locate the document. Basically, it broadcasts the request to nodes that are nearby the document that has this hash output. If the nearby nodes don't have these files, they forward the requests to their nearby nodes. This peer-finding request is quite complex. IPFS uses S/Kademlia Distributed Hash Tables, which we will discuss in a later section of this chapter.

The interesting thing is that when you use content addressing, there may be multiple providers that can serve this document. In the case of the `cute_panda.png` document, there could be more than four nodes that can serve this document. We can pick the nearest node to make the download process more efficient. This property also makes censorship much more difficult. In the case of HTTP, an actor could ban the server `https://example.com`. In the case of IPFS, however, anyone could launch a new node and serve the document. Right now, IPFS is transparent, perhaps too much so. The node that requests the document can see the IP address of the node serving the document, and vice versa. The actor could ban the IP address to forbid this document from being spread. The development to make IPFS work with Tor, software that allows users to browse websites anonymously, however, is still in its early days.

If you download a document from `https://example.com/cute_panda.png`, the document that you get at that moment may be different to the document that your friend downloaded from the same URL yesterday. It could be that the admin of the server changed the document before you downloaded it today.

With the content addressing system, however, the document that you get from the IPFS hash link, `ipfs://QmYeAiiK1UfB8MGLRefok1N7vBTyX8hGPuMXZ4Xq1DPyt7`, will always be the same, no matter when or where you download it. This hash link guarantees that nobody can tamper with the document. If you change the document and upload it to IPFS, the IPFS URL or hash would be different.

We can create a simple Python script to illustrate this case. Create a directory called `ipfs_tutorial`. Create three sample files in this directory. The first sample file is `hello.txt`, which has the content `I am a good boy.\n`. The second sample file is `hello2.txt`, which has the content `I am a good girl.\n`. The third sample file is `hello3.txt`, which has the content `I am a good horse.\n`. The fourth sample file is `hello4.txt`, which has the content `I am a good girl.\n`. The fact that the second and fourth files have the same content is deliberate. You can create different files, if you wish, but make sure that at least two of them have the same content.

Create a Python script as shown in the following code block and name it `create_hash_from_content.py`:

```
from os import listdir
from hashlib import sha256
files = [f for f in listdir('.') if 'hello' in f]
hashes = {}
for file in files:
    with open(file) as f:
        content = f.read().encode('utf-8')
        hash_of_content = sha256(content).hexdigest()
        hashes[hash_of_content] = content
content =
hashes['20c38a7a55fc8a8e7f45fde7247a0436d97826c20c5e7f8c978e6d59fa
895fd2']
print(content.decode('utf-8'))
print(len(hashes))
```

This script lists all files in the same directory that have a name that starts with `hello`. You can modify this part if your sample files don't start with `hello`. The long hash is the hash of the content of `hello2.txt`.

When you run the script, you will get the following result:

```
I am a good girl.
3
```

As you can see, there are four files, but the final output is three, not four. This is because there are three files with unique content, not four. This is how content addressing works. It does not care about the file name, it only cares about the content. It doesn't matter whether the file is called `hello1.txt` or `hello2.txt` or `hello4.txt`, it only matters that the content, `I am a good girl`.\n, is the same. Technically speaking, this is a white lie; there is a situation when IPFS must consider the filename and cannot ignore it. I'll explain the truth of this matter later in this chapter.

What we have seen in the preceding example is normal hashing. There is no Markle DAG or even Merkle tree. Let's now create a more complicated scene with a big file. Hashing a big file is not efficient. Usually, we split the file into multiple smaller pieces of the same size. For example, a 900 KB file would turn into four files. The first, second, and third files would have a size of 250 KB. The fourth file would have a size of 150 KB. Then, we hash each smaller file and combine it with a Merkle tree.

For illustration purposes, we won't use a large file, but we will impose some imaginary limitations. We don't want to hash content that spans more than one line. If the text file has four lines, we would split them into four smaller files.

Inside your project directory, create a file called `hello_big.txt` and enter the following lines:

```
I am a big boy.
I am a tall girl.
I am a fast horse.
I am a slow dragon.
```

Before we create a script to hash this big file, let's create a very simple Merkle tree library and name it `merkle_tree.py`. Refer to the GitHub link for the complete code file: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_9`.

Let's discuss this Merkle tree library, starting from its initialization:

```
def __init__(self, leaf_nodes : List[str]):
    self.hash_nodes : List[str] = []
    self.leaf_nodes : List[str] = leaf_nodes
    self._turn_leaf_nodes_to_hash_nodes()
    if len(leaf_nodes) < 4:
        self.root_hash = self._hash_list()
    else:
        self.root_hash = self._build_root_hash()
```

We make sure there are at least four nodes. If not, we might as well use the hash list technique. The `leaf_nodes` are original data nodes. They are string lists, such as `['cat', 'dog', 'unicorn', 'elephant']`. The `hash_nodes` are the hash list of the data nodes, such as `[hash of 'cat', hash of 'dog', hash of 'unicorn', hash of 'elephant']` or `['77af778...', 'cd6357e...', 'c6cb50e...', 'cd08c4c...']`.

We use the `_hash_list()` method to hash list the data if there are less than four nodes. We concatenate all the pieces of data before hashing them:

```
def _hash_list(self):
    long_node = "".join(self.hash_nodes)
    return self._hash(long_node.encode('utf-8'))
```

In the `_turn_leaf_nodes_to_hash_nodes()` method, we fill the `hash_nodes` based on the `leaf_nodes`. This is one-to-one mapping:

```
def _turn_leaf_nodes_to_hash_nodes(self):
    for node in self.leaf_nodes:
        self.hash_nodes.append(self._hash(node.encode('utf-8')))
```

In the `_hash()` method, we wrap the `sha256` hashing function. This is to make the customization of the class easier, since we may want to use a different hashing function:

```
def _hash(self, data : bytes) > bytes:
    return sha256(data).hexdigest()
```

The following code block shows how we can get the root nodes from the hash nodes:

```
def _build_root_hash(self) > bytes:
    parent_amount = ceil(len(self.hash_nodes) / 2)
    nodes : List[str] = self.hash_nodes
    while parent_amount > 1:
        parents : List[bytes] = []
        i = 0
        while i < len(nodes):
            node1 = nodes[i]
            if i + 1 >= len(nodes):
                node2 = None
            else:
                node2 = nodes[i+1]
            parents.append(self._convert_parent_from_two_nodes(node1,
node2))
            i += 2
        parent_amount = len(parents)
        nodes = parents
    return parents[0]
```

Here, we are carrying out multiple iterations on hash nodes. It jumps two steps on each iteration. For each iteration, it works on two nodes. It concatenates the hash of these two nodes, then hashes the result. The resulting hash is the parent of these two nodes. This parent becomes part of the hash nodes that will be iterated over again. This parent, along with its neighbor, will be concatenated again before being hashed, and so on. If there is an odd number of hash nodes, the last node will be concatenated with itself before being hashed. If there is only one parent, we return the hash of that, which is the root hash, as shown in the following code:

```
def _convert_parent_from_two_nodes(self, node1 : bytes, node2) ->
bytes:
    if node2 == None:
        return self._hash((node1 + node1).encode('utf-8'))
    return self._hash((node1 + node2).encode('utf-8'))
```

The `_convert_parent_from_two_nodes()` method allows us to get the parent hash from the two child nodes. We concatenate the two nodes and hash them. If the second node is `None`, meaning there is an odd number of nodes or we are processing the last node, we just concatenate the node with itself before hashing it.

Now that the Merkle tree library is ready, we will create a Python script to hash the `hello_big.txt` file and name it `hash_big_file.py then add the following code to it:`

```
from os import listdir
from hashlib import sha256
from merkle_tree import MerkleTree


hashes = {}

file = 'hello_big.txt'
with open(file) as f:
    lines = f.read().split('\n')
    hash = []
    hash_of_hash = []
    merkle_tree = MerkleTree(lines)
    root_hash = merkle_tree.root_hash

hashes[root_hash] = []
for line in lines:
    hashes[root_hash].append(line)

print(hashes)
```

If you execute this Python script, you will get the following output:

```
{'ba7a7738a34a0e60ef9663c669a7fac406ae9f84441df2b5ade3de1067c41808':
['I am a big boy.', 'I am a tall girl.', 'I am a fast horse.', 'I am a
slow dragon.', '']}
```

If the file is big, you would not hash it directly, because this could cause you to run out of memory. Instead, you split the file. Here, we split the text file based on the new lines. If you handle the binary file, you read the file chunk by chunk and save that chunk into a smaller file. Of course, before feeding them into a Merkle tree, you need to serialize the binary data into the text data. Once you have done that, you can feed the pieces of data into a Merkle tree. You get the root hash, which will protect the integrity of the original file. If you alter a single bit in a piece of data, the root hash would be different.

The Merkle DAG data structure

We have used content addressing to handle a file. If the file is big, we can split it and get the root hash with a Merkle tree. In this case, we only care about the content of the file; we don't even save its name.

There is a situation, however, where the name of the file does matter. For example, let's say that you want to save a file directory that contains 100 images of cute pandas. The names of the files in this case don't matter; what we care about is the content, the pictures of the cute pandas! If this is a directory of a programming project, the names of the files do matter. If one Python file tries to import another Python library that is contained in a different file, we have to keep the name of the file. Let's say that we have a Python file called `main.py` that has the following content:

```
from secret_algorithm import SuperSecretAlgorithm
# execute it
SuperSecretAlgorithm()
```

The `main.py` file is dependent on another file in the same directory called `secret_algorithm.py`. It is not just the content of the `secret_algorithm.py` file that matters, but also its name. If the filename changes, `main.py` will not be able to import the library.

In order to save the content and the filename, we need to use a Merkle DAG data structure. As mentioned before, one of the differences between a Merkle DAG and a Merkle tree is that any node in a Merkle DAG can hold data, not just a leaf node, as is the case in a Merkle tree.

Let's create a sample directory that contains sample files and a nested directory that also contains files:

```
$ mkdir sample_directory
$ cd sample_directory
$ // Create some files
$ mkdir inner_directory
$ cd inner_directory
$ // Create some files
```

Then, create a Python script to explain this new data structure. Create a file called `merkle_dag.py` in your project directory. Refer to the GitHub link for the complete code file: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_10`.

Let's discuss the `MerkleDAGNode` class, starting from its initialization method:

```
    def __init__(self, filepath : str):
        self.pointers = {}
        self.dirtype = isdir(filepath)
        self.filename = Path(filepath).name
        if not self.dirtype:
            with open(filepath) as f:
                self.content = f.read()
            self.hash = self._hash((self.filename + self.content).
encode('utf-8'))
        else:
            self.content = self._iterate_directory_contents(filepath)
```

```
            nodes_in_str_array = list(map(lambda x: str(x), self.
content))
            if nodes_in_str_array:
                self.hash = self._hash((self.filename +
MerkleTree(nodes_in_str_array).root_hash).encode('utf-8'))
            else:
                self.hash = self._hash(self.filename.encode('utf-8'))
```

The `__init__()` method accepts a file path as an argument. This could be a path to a file or a directory. We make an assumption that this is a valid path and not a symbolic link. `self.pointers` will be explained later on in the section with the `_iterate_directory_contents()` method. `self.dirtype` is used to differentiate between the directory or the file. `self.filename` is used to hold the name of the file or the name of the directory.

If the argument is the path to the file (not the directory), we read the content into `self.content`. For demonstration purposes, we assume the content of the file is small and we don't try to split the files like we did before. Then, we calculate the hash based on the filename and the content.

If the argument is the path to the directory, the content would be an array of `MerkleDAGNode` objects of the inner files inside that directory. To calculate the hash, we use a Merkle tree to get the root hash of its children. However, we need to concatenate this with the name of the directory before hashing it again as shown in the following code:

```
    def _hash(self, data : bytes) -> bytes:
        return sha256(data).hexdigest()
```

`_hash()` is a wrapper method of the sha256 hashing function. The `_iterate_directory_contents()` method is used to iterate over the inner children of the directory. We convert every file or directory inside this directory to a `MerkleDAGNode` object. The `self.pointers` object is used to make it easier to access the `MerkleDAGNode` based on the filename. Basically, it is like a recursive function, especially when we hit a directory as shown in the following code:

```
    def _iterate_directory_contents(self, directory : str):
        nodes = []
        for f in listdir(directory):
            merkle_dag_node = MerkleDAGNode(directory + '/' + f)
            nodes.append(merkle_dag_node)
            self.pointers[f] = merkle_dag_node
        return nodes
```

The `_repr_()` method is used to make it easier to print objects for debugging as shown in the following code:

```
    def __repr__(self):
        return 'MerkleDAGNode: ' + self.hash + ' || ' + self.filename
```

The `__eq__()` method is needed so that we can compare the `MerkleDAGNode` object with other `MerkleDAGNode` objects. This is useful during the testing process as shown in the following code:

```
    def __eq__(self, other):
        if isinstance(other, MerkleDAGNode):
            return self.hash == other.hash
        return False
```

Let's create a `hash_directory.py` file to demonstrate the power of this data structure as shown in the following code:

```
from merkle_dag import MerkleDAGNode

outer_directory = 'sample_directory'

node = MerkleDAGNode(outer_directory)
print(node)
print(node.content)
```

You would get the following result if you execute the script:

```
MerkleDAGNode:
b2305221b5fa3c780433fab01015e4a9293b046097f20240fcdf5d5661022b31 ||
sample_directory
[MerkleDAGNode:
8ced218a323755a7d4969187449177bb2338658c354c7174e21285b579ae2bca
|| hello.txt, MerkleDAGNode:
bc908dfb86941536321338ff8dab1698db0e65f6b967a89bb79f5101d56e1d51
|| hello3.txt, MerkleDAGNode:
c075280aef64223bd38b1bed1017599852180a37baa0eacce28bb92ac5492eb9
|| inner_directory, MerkleDAGNode:
97b97507c37bd205aa15073fb65367b45eb11a975fe78cd548916f5a3da9692a ||
hello2.txt]
```

The output is the schema of the Merkle DAG node.

This is how Git keeps the files. Our implementation is just for understanding purposes and would not be fit for production purposes. In the real world, you should have many optimizations. One of the optimizations that you could implement is using a reference for the data, just like Git. If there are two different files that have the same content (but different filenames), the content would be saved just once. The other optimization is that Git uses compression. *Figure 9.8* illustrates the concept of Git, where we have two files, *file B* and *file D*. See *Figure 9.8*:

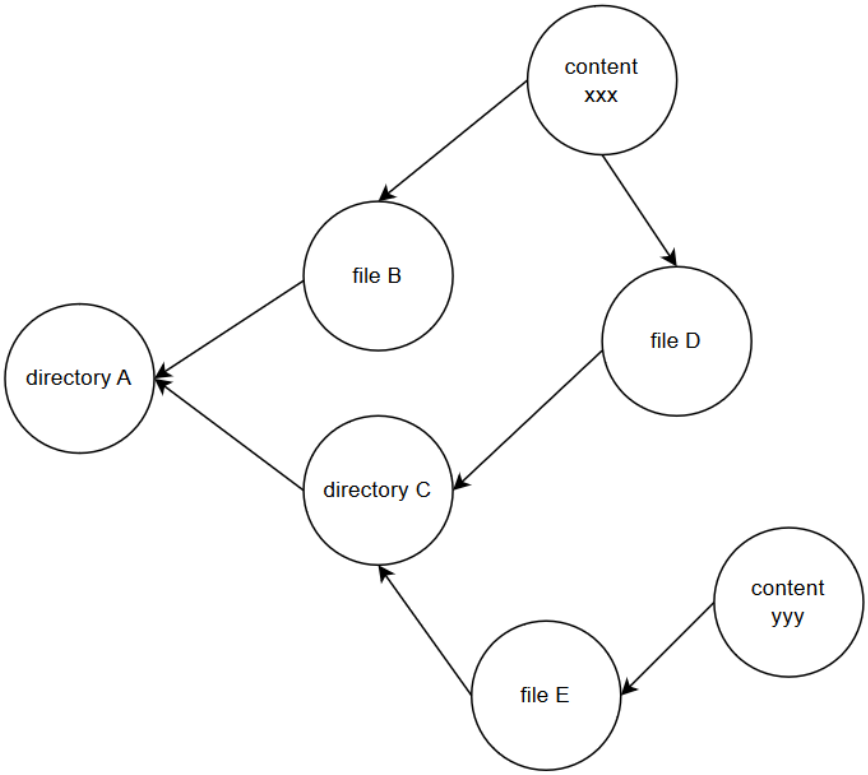


Figure 9.8: How Git keeps files

As shown in *Figure 9.8*, these both have the same content, *content xxx*. File B is saved just once in directory A. File D is saved at directory C with file E, which has a different content, content yyy. Directory C is also saved in directory A. But the content of File B and File D, which is content xxx, is saved only once.

Now that we know how to save a directory of files with Merkle DAG, what if we want to change the content of the file? Should we abandon this Merkle DAG node and create a totally new node? A more efficient way to solve this problem would be to use a versioning system. A file could have version 1, version 2, version 3, and so on. The easiest way to implement versioning is to use a linked list, as illustrated in the following diagram. See *Figure 9.9*:

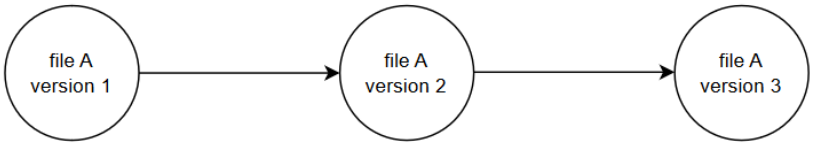


Figure 9.9: Version of a file as a linked list

Now that you understand Merkle DAG which is the basic data structure of IPFS, you can move to the networking side.

Peer-to-peer networking

We understand how to save files in IPFS. *The key is the hash*. The value is the name of the file or directory and the content of the file or directory. If we were building a centralized system, our story would be finished. We would just need to add a few other things to create a piece of software to save files and search them based on the hash. This software would be similar to a database, such as SQLite or LevelDB. IPFS is neither of those; it is a peer-to-peer filesystem that is like a database but spread all over the place. In other words, it is a distributed hash table.

IPFS uses S/Kademlia, an extended version of Kademlia, which is a distributed hash table. Before we discuss Kademlia, let's discuss its predecessor.

First, imagine a hash table, which is like a dictionary data structure in Python. The dictionary has two parts, the key and the value, as shown in the following table:

Key	Value
2	Cat
5	Unicorn
9	Elephant
11	Horse
4	Rhino
101	Blue Parrot
33	Dragon

Table 9.1: The sample table of animals

In IPFS, the key is the hash, not a number. But for demonstration purposes, let's make it a simple integer. The value is just a simple name of animal, not the content of the file or the content of the files inside a directory.

Now, imagine you have four nodes. A node could be a computer that is located in a different continent to the rest of the nodes.

Let's define which node holds which keys. See *Table 9.2*:

Node	Keys
A	2, 9, 11
B	5
C	4, 33
D	101

Table 9.2: The sample table of numbers

You keep this table in a central server. One of the nodes will be the central node. This means that if someone wants to access key five, they have to ask the central server before receiving the answer, node B. After that, the request can be directed to node B. Node B would return "Unicorn" to the data requester.

This method is very efficient; no time is wasted. Napster, the peer-to-peer music sharing system, uses this approach. The drawback is that the central server is a single point of failure. An adversary (someone who does not like this information being spread; in the case of Napster, this could be a big music labels) could attack the central server.

One solution would be to ask all nodes about which node holds the key instead of keeping this information in the central node. This is what Gnutella does. This setup is resilient to censorship and attacks from adversaries, but it makes life hard for nodes and people who request the data. The node must work hard when receiving many requests. This setup is called flooding. It is suitable for Bitcoin, but not for IPFS.

This is why the distributed hash table technology was created. There are a couple of distributed hash table algorithms, one of which is **Kademlia**. This algorithm was created by Petar Maymounkov and David Mazières in 2002. It was later used by the eDonkey file sharing platform.

The notion of proximity of data and nodes

In a distributed hash table, we don't put the data in every node. We put the data in certain nodes according to the notion of closeness. We want to put the data in nearby nodes. This means that we have the concept of distance not just between nodes, but also between the data and the nodes.

Imagine that every node launched or created in this distributed hash table is given an ID between 1 and 1000. Every node ID is unique, so there can be a maximum of 1,000 nodes. There are likely to be more than 1,000 nodes in a real-world setting, but this will work as an example. Let's say that we have 10 nodes. See *Table 9.3*:

Node ID
5
13
45
48
53
60
102
120
160
220

Table 9.3: Node IDs

We also have some data. To make it simple, the data in this case is just some strings. See *Table 9.4*:

Data
Unicorn
Pegasus
Cat
Donkey
Horse

Table 9.4: Data of animals

To be able to say whether this data is close to or far from certain nodes, we need to

convert this data into a number between 1 and 1000. In the real world, you could hash the data. But for our practical demonstration, we will just allocate a random number. See *Table 9.5*:

Key	Data
54	Unicorn
2	Pegasus
100	Cat
900	Donkey
255	Horse

Table 9.5: Hash table of animals

If we want to store the Unicorn data in the four nearest nodes (four is just a configuration number), this can be done as follows. First, you check the key, which is 54. Then, we want to get the nearest four nodes to 54. If you check the node ID list, the nearest four nodes are 45, 48, 53, and 60. So, we store the Unicorn data in these four nodes. If we want to store the Cat data, the nearest neighbors from its key, 100, are 53, 60, 102, and 120, so we store the Cat data in these four nodes.

We treat data as a node when calculating the distance. This is how we look up data in a distributed hash table. The data and the nodes share the same space.

XOR distance

However, in Kademlia, we don't measure distance by decimal subtraction. To make it clear, decimal subtraction is just normal subtraction. The distance between 45 and 50 is 5. The distance between 53 and 63 is 10.

In Kademlia, measuring distance is done by XOR distance. The XOR distance between 3 and 6 is 5, not 3. Here's how to count it:

The binary version of 3 is 011. The binary version of 6 is 110. What I mean by binary version is the number in base 2. XOR means exclusive or. Using the XOR operation, 1 XOR 0 is 1, 1 XOR 1 is 0, 0 XOR 0 is 0, and 0 XOR 1 is 1. If two operands are same, the result is 0. If two operands are different, the result is 1, as show in the following operation:

```
011
110
---xor
101
```

101 is the binary version of 5.

The XOR distance has a few useful properties that prompted the author of the Kademlia paper to choose the XOR distance to measure the distance between the nodes.

The first property is that the XOR distance of a node to itself is 0. The closest node to a node with an ID of 5 is another node with an ID 5, or itself. The binary version of 5 is 0101. See the following operation:

```
0101
0101
----xor
0000
```

The 0 distance is only possible if we measure the distance between a node and itself.

The second property is that the distance between different nodes is symmetrical. The XOR distance between 4 and 8 is same as the XOR distance between 8 and 4. The binary version of 4 is 0100 and the binary version of 8 is 1000. So, if we calculate the distance between them using their binary value, we get the same value. The XOR distance between 4 and 8 is as follows:

```
0100
1000
----xor
1100
The XOR distance between 8 and 4 is as follows:
1000
0100
----xor
1100
```

If you are used to working with decimal subtraction distances, this will be intuitive to you.

The last useful property is that the distance between node X and node Z is less than or equal to the distance between node X and node Y plus the distance between node Y and Z. This last property is important because a node in a Kademlia distributed hash table does not save all the other nodes' addresses. It only saves some nodes' addresses. But a node can reach another node through intermediate nodes. Node X knows the address of node Y, but does not know the address of node Z. Node Y does know the address of node Z. Node X can query the neighbor nodes of node Y from node Y. Then, node X can reach node Z knowing that the distance to node Z is less than or equal to the distance of node X and node Y added to the distance of node Y and node Z.

If this property were not true, the longer node X searches for a node, the further the distance a particular node will be, which is not what we wanted. But with this property, the addresses of neighbor nodes from other nodes may be smaller than, if not the same as, the combined distances.

When you think about using XOR distance, you should think that the more prefixes shared by two numbers, the shorter the distance between those two numbers. For example, if the numbers share three common prefixes, such as five and four, the distance is one, as you can see in the following operation:

```
0101
0100
----xor
0001
```

Likewise, for numbers 14 and 15, the distance is also 1, as you can see in the following operation:

```
1110
1111
----xor
0001
```

But, if the bit differences are on the left side, such as is the case for 5 and 13, the distance might be large, in this case eight, as you can see in the following operation:

```
0101
1101
----xor
1000
```

The XOR distance between 4 and 5 is 1 but the XOR distance between 5 and 6 is 3. This is counter-intuitive if you are accustomed to decimal subtraction distances. To make this concept easier to explain, let's create a binary tree that is composed of numbers from 1 to 15. See *Figure 9.10*:

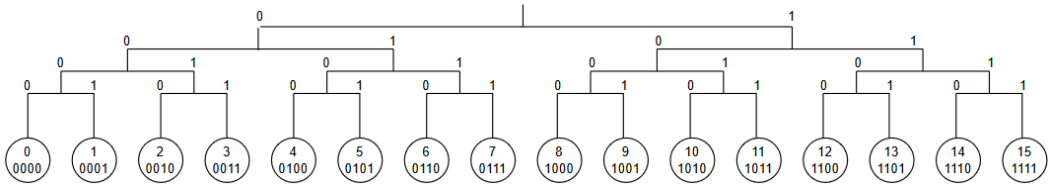


Figure 9.10: Binary tree

Look at this tree carefully. The XOR distance between 4 and 5 is 1, but the XOR distance between 5 and 6 is 3. If you look at the picture, 4 and 5 are under an immediate branch, whereas 5 and 6 is under a larger branch, which implies a larger distance. The immediate branch corresponds to the bit on the right. The parent branch of that immediate branch corresponds to the second-most right bit. The top branch corresponds to the bit on the left. So, if the number is separated by a top branch, the distance is at least 8. The binary version of 8 is 1000.

This is just for understanding purposes; it is not a rigorous mathematical definition. If you look at the journey from 5 to 11 and 5 to 13, you should get roughly the same distance, but this is not the case. The XOR distance of 5 and 13 is 8 but the XOR distance of 5 and 11 is 14.

In Python, you can XOR two numbers with the `^` operator:

```
>> 5 ^ 11
14
```

You can turn any decimal number to its binary version using the `bin` function:

```
>>> bin(5)
'0b101'
```

Then, if you want to convert the binary number back to a decimal number, use the `int`

function:

```
>>> int('0b101', 2)
5
```

The second argument of the `int` function indicates which base the first argument is. Binary is base 2.

Buckets

Now that we have gone through XOR distances, we will take a look at how a node saves other nodes' addresses. A node does not save all other nodes in a distributed hash table. The number of nodes a node can save depends on the number of bits in a node and the k configuration number. Let's discuss these one by one.

Remember the tree picture we saw previously? It has 16 leaves. Now imagine the smallest tree. See *Figure 9.11*:

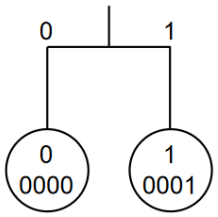


Figure 9.11: A branch

As you can see, this tree has two leaves. Let's double the tree. See *Figure 9.12*:

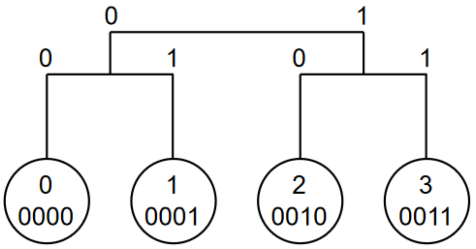


Figure 9.12: Two branches

The tree now has four leaves. Let's double it again. See *Figure 9.13*:

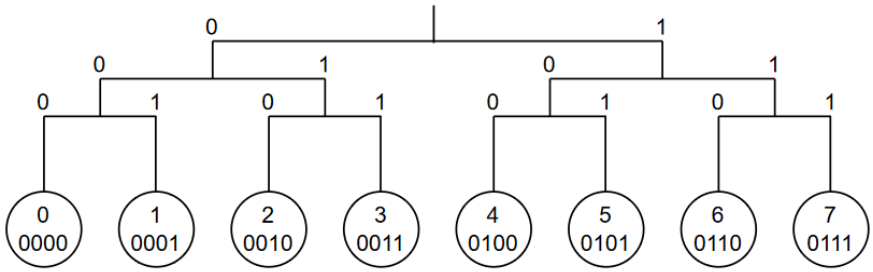


Figure 9.13: Four branches

The tree now has eight leaves. If you double it again, you would have a tree like our

previous tree, which has 16 leaves.

The progression we can see is 2, 4, 8, 16. If we continue the journey, the numbers would be 32, 64, 128, and so on. This can be written as 2 1, 2 2, 2 3, 2 4, 2 5 ... 2 n .

Let's focus on the tree with 16 leaves. When we represent the leaf number, we must use a 4- bit binary number, such as `0001` or `0101`, because the biggest number is 15, or 1111. If we use a tree with 64 leaves, we must use a 6-bit number, such as `000001`, 010101 because the biggest possible number is `63` or `111111`. The bigger the bit number, the larger the amount of nodes a node must save in its address book.

Then, we have the `k` configuration number. `k` decides the maximum amount of nodes a node can save in a bucket. The number of buckets is the same as the number of bits used in a distributed hash table. In a tree with 16 leaves, the number of buckets is 4. In a tree with 64 leaves, the number of buckets is 6. Each bucket corresponds to a bit. Let's say we have a tree with 16 leaves, so each number has 4 bits, such as 0101 or 1100. This means the node has four buckets.

The first bucket corresponds to the first bit from the left. The second bucket corresponds to the second bit from the left. The third bucket corresponds to the third bit from the left. The fourth bucket corresponds to the fourth bit from the left.

Let's take a look at the example of a node with ID 3 in a tree with 16 leaves. For now, we assume we have 16 nodes in a tree that has 16 leaves. In the real world, the tree would be sparse and a lot of branches would be empty.

In the paper that describes Kademlia, the authors used 160 buckets or a 160-bit address. The number of leaves in this tree is vast. For comparison, is the number of atoms in visible universe. The k configuration number is chosen as 20 in this paper, so a node can have a maximum of 3,200 nodes in its address book.

For this example, let's say that the k number is 2. This means for every bucket, the node saves two other nodes. The first bucket, which corresponds to the first bit, corresponds to the other half of the tree, where the node does not reside. We have eight nodes in this half of the tree but we can only save two of them because the k number is 2. Let's choose nodes 11 and 14 for this bucket. How we choose which nodes go in which buckets will be described later.

How Kademlia works in a nutshell:

Then, let's divide the half of the tree where the node resides, so we have two branches. The first branch consists of a node with ID 0, a node with ID 1, a node with ID 2, and a node with ID 3. The second branch consists of a node with ID 4, a node with ID 5, a node with ID 6, and a node with ID 7. This second branch is the second bucket. There are four nodes in this branch, but we can only save two nodes. Let's choose the node with ID 4 and the node with ID 5.

Then, let's divide the branch where our node (the node with ID 3) resides so we have two small branches. The first small branch consists of a node with ID 0 and a node with ID 1. The second small branch consists of a node with ID 2 and a node with ID 3. So the third bucket is the first small branch. There are only two nodes, a node with ID 0 and node with ID 1, so we save both.

Finally, let's divide the small branch where our node (the node with ID 3) resides so we have two tiny branches. The first branch consists of a node with ID 2 and the second branch consists of a node with ID 3. The fourth bucket, or the last bucket, would be the branch that consists of node 3.

We save this one node because it is less than the k configuration number. See *Figure 9.14*:

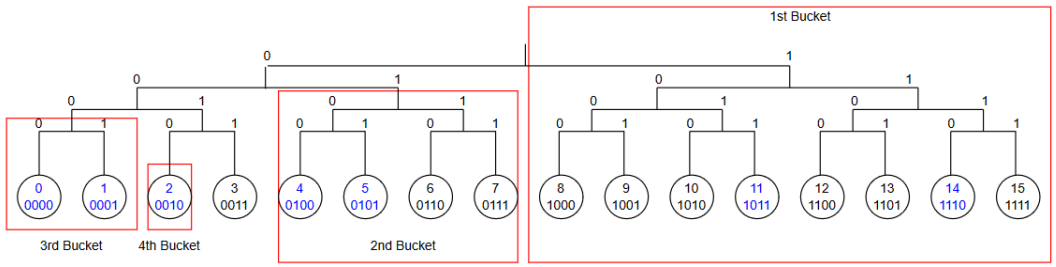


Figure 9.14: Diving a binary tree to buckets

The following diagram shows the full four buckets. Each bucket is half of the branch in which the source node does not reside. The bucket configuration of different nodes are different. The node with ID 11 could have a bucket configuration that looks as follows. See *Figure 9.15*:

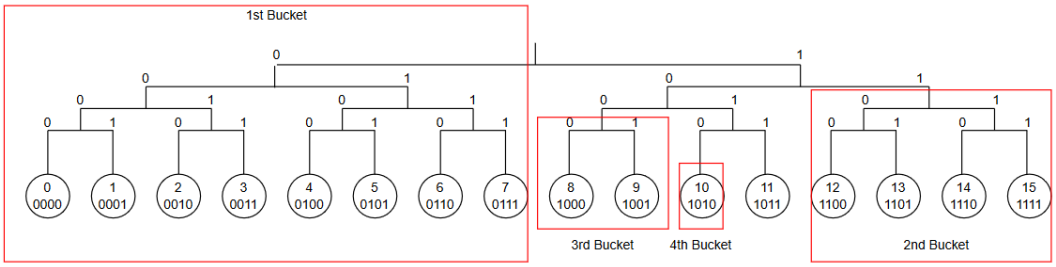


Figure 9.15: The four full buckets

Let's take a look at an example of how a certain node could find another node that does not reside in its address book. Imagine the k configuration number is 1. The source node is the node with the ID 3 in a tree with 16 leaves. For the first bucket (the largest branch that consists of the nodes from ID 8 to ID 15), the node with ID 3 saves the node with ID 10. But the node with ID 3 wants to find the node with ID 13. The node with ID 3 contacts the node with ID 10 with a request, "Can you help me find the node with ID 13?". The node with ID 10 has saved the node with ID 14 in its corresponding bucket (the branch that consists of nodes with IDs 12, 13, 14, and 15). The node with ID 10 gives the node with ID 14 to the node with ID 3. The node with ID 3 asks the same question to the node with ID 14, "Can you help me find the node with ID 13?". The node with ID 14 does not have it, but it has the node with ID 12 in its bucket (the branch that consists of the node with ID 12 and the node with ID 13). The node with ID 14 gives the node with ID 12 to the node with ID 3. The node with ID 3 asks the same question again to the node with ID 12. This time, the node with ID 12 can give the destination node or the node with ID 13 to the node with ID 3. This brings us to a happy ending!

The following diagram shows the nodes. See *Figure 9.16*:

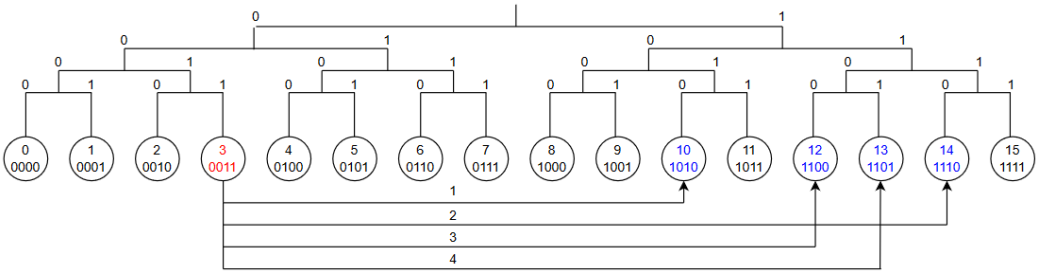


Figure 9.16: A node finding other nodes

Did you notice how many times the node ID 3 must repeat the request? Four times. If this number sounds familiar, that is because this tree has 16 leaves, which is 2 4. In computer science, the worst case scenario of the amount of hopping required before getting to the destination is 2 log n + c. n is how many leaves the tree has and c is constant number.

The tree you have just seen has full nodes; there are no empty leaves or empty branches. In the real world, however, there are empty branches and empty leaves. Imagine that you have a tree with 1,024 (2^{10}) leaves and the k number is 3. You launch the first node with the ID 0. This node will be the source node. We will see the tree from the lens of the node with ID 0. See *Figure 9.17*:

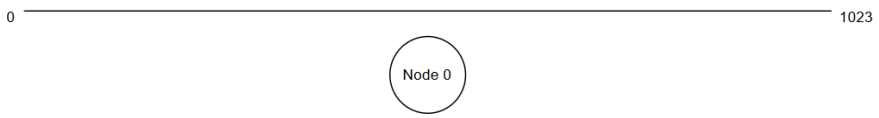


Figure 9.17: Node 0

Then, you launch the node with ID 800. See *Figure 9.18*:

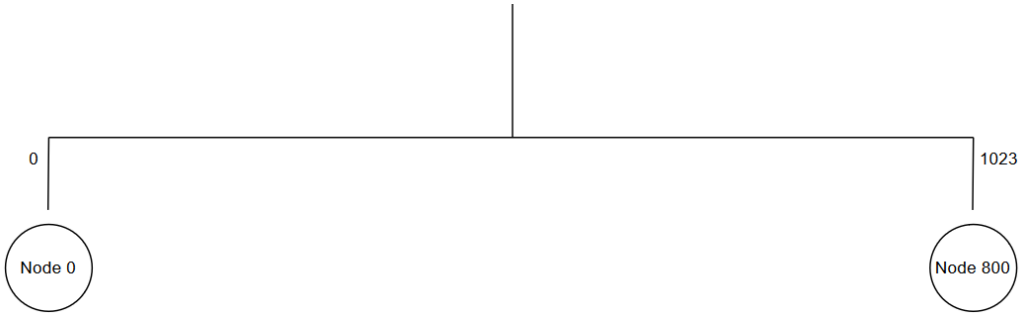


Figure 9.18: Two nodes

The tree will be split into two buckets. Then, you launch the node with ID 900 and the node with ID 754. See *Figure 9.19*:

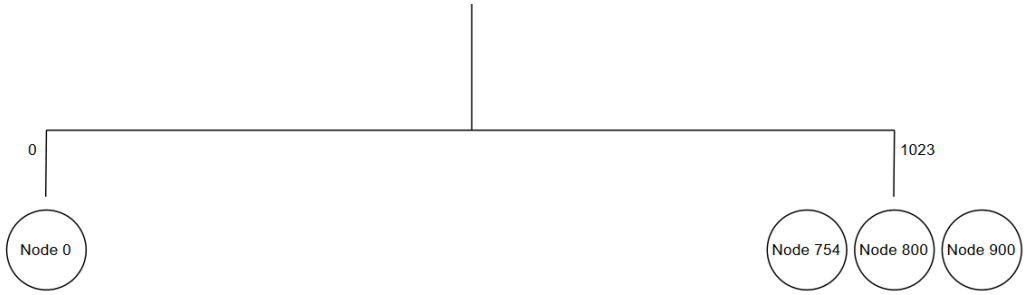


Figure 9.19: Four nodes

What if we add another node to the bucket? Let's launch the node with ID 1011. The node with ID 0 will ping the least recently used node, which is the node with ID 800, to see if it is still alive. If it is, it will check the other nodes. If the node with ID 754 is not alive, then this node will be replaced with the node with ID 1011. If all the nodes are still alive, then the node with ID 1011 will be rejected from the bucket. The reason for this is to avoid new nodes swamping the system. We assume that the nodes with longer uptimes are trustworthy and we prefer these nodes to new nodes. Let's say we reject the node with ID 1011.

First, we launch the node with ID 490. Then, we split the branch where the node with ID 0 resides. See *Figure 9.20*:

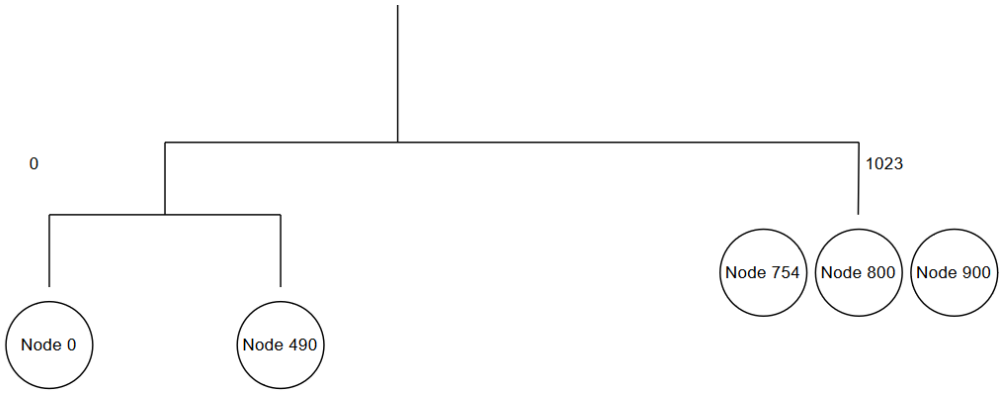


Figure 9.20: Splitting branches

Now, let's add the node with ID 230. See *Figure 9.21*:

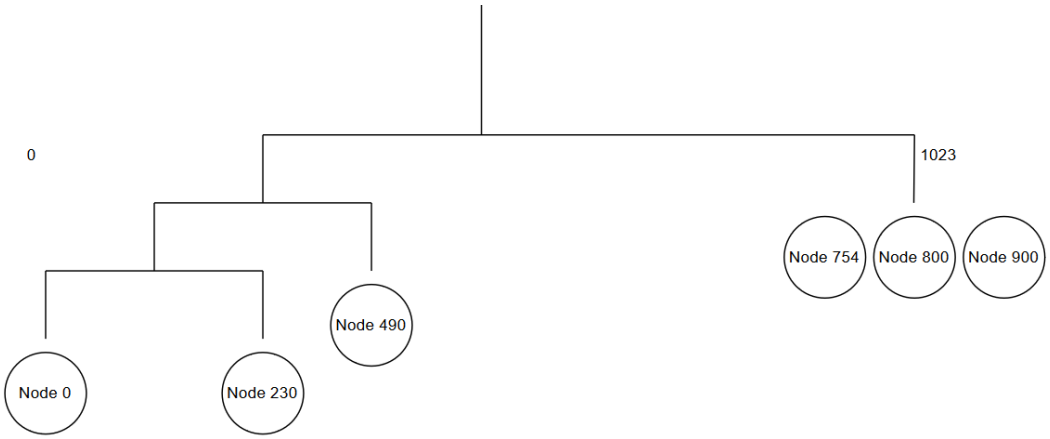


Figure 9.21: Splitting branches again

Let's add the node with ID 60. See *Figure 9.22*:

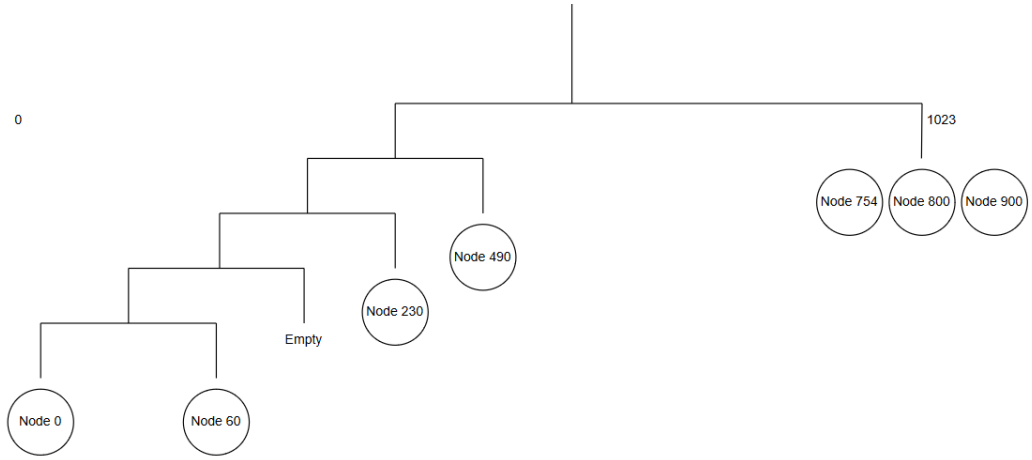


Figure 9.22: Many branches

If we keep adding nodes, the same process happens again, and again. Every time we add a node in a branch where the source node resides, it will split the bucket into two until it reaches the lowest level. If we add a node in other branch on which the source node does not live, we add nodes until we reach the k number.

You now have a basic understanding of how Kademlia works. This is not, however, the whole story. If a node is inserted, a node needs to tell the older nodes of its existence. That node also needs to get the contacts from the old node. I mentioned that the branch is split when a node is inserted to a branch on which the source node resides, but there is a case where the branch is split even when the source node does not reside there. This happens because a node is required to keep all valid nodes in a branch that has at least k nodes if that means the branch in which the source node does not reside has to be split.

There are other important aspects of Kademlia other than routing algorithms. A node is required to republish the key and the value (the data) every hour. This is to anticipate the old nodes leaving and the new nodes joining the system. These nodes are nearer, so they are more suited to keep the data. There is also an accelerated lookup algorithm so that we can use fewer steps when a node is looking for another node.

Note

You can refer to the Kademlia paper for the full specification. `https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf`.

IPFS uses `S/Kademlia`, an extended version of Kademlia. It differs from the original Kademlia algorithm in that S/Kademlia has some security requirements. Not all nodes join the Kademlia distributed hash table with a noble purpose. So, in S/Kademlia, to generate the ID of a node, it requires the node to generate a cryptography key pair, so it is very difficult to tamper with the communication between the nodes. Other requirements include the fact that proof-of-work (like in Bitcoin and Ethereum) is used before a node is able to generate its ID. There is also some adjustment in the routing algorithm to make sure a node can communicate with other nodes in the midst of adversaries, such as nodes that spam the network.

Summary

In this chapter, we have studied IPFS. We started by looking at the motivations of the IPFS project and its history. Although IPFS is not a part of the blockchain technology, it is similar to blockchain because it complements blockchain technology. We then learned about the data structure of the content that we saved in the IPFS filesystem. This data structure is Merkle Directed Acyclic Graph (DAG), which is based on the Merkle tree. We created simple Merkle tree and Merkle DAG libraries to understand the uniqueness of these data structures. Merkle trees provide an easy way to check the integrity of partial data, while Merkle DAGs are used when we want to save a directory with files and we want to keep the filenames. Then, we learned about the peer-to-peer networking aspect of a Kademlia distributed hash table. The distance between nodes is based on the XOR distance. The nodes also are kept in buckets, which corresponds to bit addressing. Finally, we showed how a node can find other nodes by hopping through the buckets. With this knowledge, we can understand and use the technology of the decentralized storage that fixes the weakness of the blockchain technology.

In the next chapter, we are going to create an application that uses the IPFS software.

10

Implementing a Decentralized Application Using IPFS

In this chapter, we are going to combine a smart contract with the **InterPlanetary File System** (**IPFS**) to build a decentralized video-sharing application (similar to YouTube but decentralized). We will use a web application as the front end for the blockchain and IPFS. As stated previously, IPFS is not a blockchain technology – it is a decentralized technology. However, in a blockchain forum, meetup, or tutorial, you may hear IPFS being mentioned quite often. One of the main reasons for this is that IPFS overcomes the weakness of blockchain, which is that its storage is very expensive. After finishing this chapter, you will be able to create a blockchain application that still maintains the advantages of blockchain (transparency, censorship resistance, and so on) while using storage. You'll understand how to link between data in storage and values in smart contracts.

In this chapter, we will cover the following topics:

- Installing the IPFS software and its library
- The architecture of the decentralized video-sharing application
- Writing the video-sharing smart contract
- Building the video-sharing web application

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_10`.

Installing the IPFS software and its library

This application that you're going to build needs IPFS to store videos. So, we'll start this chapter by installing the IPFS application. Go to `https://github.com/ipfs/kubo/releases/` and download the latest version, which is 0.28.0 at the time of writing. You'll need to use the `kubo_v0.28.0_linux-amd64.tar.gz` file if you're using the Linux platform. Make sure you download the application that's suitable for your operating system.

Extract the file that you downloaded previously – in my case, `kubo_v0.28.0_linux-amd64.tar.gz` – by running the following command:

```
$ tar -zxvf kubo_v0.28.0_linux-amd64.tar.gz
kubo/LICENSE
kubo/LICENSE-APACHE
kubo/LICENSE-MIT
kubo/README.md
kubo/build-log
kubo/install.sh
kubo/ipfs
```

Then, go inside the directory and run the script:

```
$ cd go-ipfs
$ sudo ./install.sh
```

This step is optional. Here, we're exporting the `IPFS_PATH` environment variable to our shell:

```
$ export IPFS_PATH=/path/to/ipfsrepo
```

This is where `ipfs` stores the files. You can store this statement in `~/.bashrc`. By default (without this environment variable), the IPFS application would use `~/.ipfs` (the `.ipfs` directory in the home directory) to store data.

After setting up the environment variable, initialize the `ipfs` local repository. You only need to perform this step once:

```
$ ipfs init
```

Then, launch the IPFS daemon, as follows:

```
$ ipfs daemon
```

By default, the API server is listening on port `5001`. We will interact with the programmatically through this port. Also, by default, it only listens in localhost. Be careful if you want to open this port to the outside world. There is no **access control list** (**ACL**) in IPFS. Anyone who has access to this port can upload data to IPFS.

Now that the IPFS daemon has been launched, let's install the `aioipfs` Python library:

```
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install aioipfs
```

The `aioipfs` library is based on the asyncio Python library, which uses asynchronous-style programming. You need to be familiar with it to use the library. In the next section, we'll explore the IPFS Python library!

Exploring the IPFS Python library

First things first – let's download a cat picture from IPFS:

```
$ ipfs cat /ipfs/QmW2WQi7j6c7UgJTarActp7tDNikE4B2qXtFCfLPdsgaTQ/cat.
jpg > cat.jpg
$ eog cat.jpg
```

When you run the preceding code, a cat picture, as shown in *Figure 10.1*, will be downloaded and you will get the following as output:

Figure 10.1: A cat picture from IPFS

Note

Eog is an image viewer in Linux. So, to view an image, I need to run the `eog` application. You can use another image viewer application if you wish. On Mac, you can use the `open` command while specifying the image's filename as an argument to view the image.

Let's create a Python script that will download the preceding image programmatically with Python. Name it `download_cute_cat_picture.py`:

```
import asyncio
import aioipfs

async def get():
    client = aioipfs.AsyncIPFS()

    cute_cat_picture =
'QmW2WQi7j6c7UgJTarActp7tDNikE4B2qXtFCfLPdsgaTQ/cat.jpg'
    await client.get(cute_cat_picture, dstdir='.')
    await client.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(get())
```

The preceding code uses the `AsyncIPFS` method from `aioipfs` to get the connection object. Then, you can use the `get` method from this connection object with the hash content and the destination directory as parameters.

Now, let's execute the script:

```
(.venv) $ python download_cat_picture.py
```

Once you've done this, the `cat.jpg` file will be in your current directory:

```
(.venv) $ ls cat.jpg
cat.jpg
```

This is the same cute cat picture you saw previously. As you may have noticed, there is a `cat.jpg` filename after the long hash. Technically speaking, what we are doing here is downloading a file inside a directory that contains a cute cat picture. You can try this if you like. To do so, create another script and name it `download_a_directory_of_cute_cat_picture.py`, then add the following code:

```
import asyncio
import aioipfs

async def get():
    client = aioipfs.AsyncIPFS()

    cute_cat_dir = 'QmW2WQi7j6c7UgJTarActp7tDNikE4B2qXtFCfLPdsgaTQ'
    await client.get(cute_cat_dir, dstdir='.')
    await client.close()
loop = asyncio.get_event_loop()
loop.run_until_complete(get())
```

After executing this script, you will get a directory named `QmW2WQi7j6c7UgJTarActp7tDNikE4B2qXtFCfLPdsgaTQ` in the directory that contains this script. If you peek inside this directory, you will find the cat picture file.

Let's take a look at the script line by line to understand the usage of the `aiopfs` library. You can use the following code to import the library:

```
import aioipfs
```

The following code is used to get a connection object to the IPFS daemon:

```
client = aioipfs.AsyncIPFS()
```

The `connect` method accepts a couple of parameters. The two most important parameters are `host` and `port`:

```
client = aioipfs.AsyncIPFS(host='localhost', port=5001)
```

By default, we connect to localhost via `port 5001`.

We can try and get the picture by running the following command:

```
await client.get(cute_cat_dir, dstdir='.')
```

We use the methods of the IPFS HTTP API from the `client` object. The `get` method is one of the methods that's used to interact with the IPFS daemon. For this method, there is usually an equivalent argument for the `ipfs` client software:

```
$ ipfs get QmW2WQi7j6c7UgJTarActp7tDNikE4B2qXtFCfLPdsgaTQ/cat.jpg
```

Now, let's add a file to IPFS. To do this, create a simple file and name it `hello.txt`. This is the content of the file:

```
I am a good unicorn.
```

Make sure you have a new line after the string:

```
$ cat hello.txt
I am a good unicorn.
$
```

If the command prompt comes after the line of the string, then all is well. You can carry on.

However, let's say the command prompt comes on the right of the string, as shown in the following code block:

```
$ cat hello.txt
I am a good unicorn.$
```

This means you don't have the new line and you need to add the new line after the string.

Now, let's create a script to add this file to IPFS. Name it `add_file.py`. Then, add the following code to it:

```
import asyncio
import aioipfs

async def add_file():
    client = aioipfs.AsyncIPFS()

    files = ['hello.txt']
    async for added_file in client.add(files):
        print('Imported file {0}, CID: {1}'.format(
            added_file['Name'], added_file['Hash']))
    await client.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(add_file())
```

Executing this code will give you the following output:

```
(.venv) $ python add_file.py
Imported file hello.txt, CID:
QmY7MiYeySnsed1Z3KxqDVYuM8pfiT5gGTqprNaNhUpZgR
```

We can retrieve the content of the file by using the `cat` or `get` method. The content is `I am a good unicorn.\n`. Notice that there is a newline character (`\n`) next to the `unicorn`. Let's use the `cat` method in the `get_unicorn.py` script, as shown in the following code:

```
import asyncio
import aioipfs

async def cat_file():
    client = aioipfs.AsyncIPFS()

    hash = 'QmY7MiYeySnsed1Z3KxqDVYuM8pfiT5gGTqprNaNhUpZgR'
    result = await client.cat(hash)
    print(result)
    await client.close()
```

```
loop = asyncio.get_event_loop()
loop.run_until_complete(cat_file())
```

Running this will give you the following output:

```
(.venv) $ python get_unicorn.py
b'I am a good unicorn.\n'
```

As we mentioned in *Chapter 9*, *InterPlanetary – a Brave New Filesystem*, we get the content of the file through the hash. Here, we only retrieve the content, not the name of the file.

Now that you know how to use the IPFS application and the IPFS Python library, let's create the architecture of the decentralized application that will use IPFS as a form of storage.

The architecture of the decentralized video-sharing application

You can imagine the video-sharing application this way: first, you go to a website, where you will see a list of videos (just like YouTube). Here, you can play videos in your browser, upload videos to your browser so that people can watch your cute cat video, and *like* other people's videos.

On the surface, this is like a normal application. You build it with your favorite Python web framework, such as Django, Flask, or Pyramid. Then, you use MySQL or PostgreSQL as the database. You could choose NGINX or Apache as the web server in front of the Gunicorn web server. For caching, you can use Varnish for full-page caching and Redis for template caching. You will also host the web application and videos on the cloud, such as via **Amazon Web Service** (**AWS**), **Google Cloud Platform** (**GCP**), or Azure. Then, you will use a content delivery network to make it scalable worldwide. For the frontend side, you could use the JavaScript framework with React.js, Angular.js, Vue.js, or Ember. If you are an advanced user, you could use machine learning for video recommendations.

However, the key point here is that what we want to build is a decentralized video-sharing application with blockchain technology, not a centralized application.

Let's discuss what we mean by building a decentralized video-sharing application with blockchain technology.

We cannot store video files on the Ethereum blockchain as it is very expensive; even storing a picture file costs an arm and a leg on the Ethereum blockchain. Someone has done the math on this for us at `https://ethereum.stackexchange.com/questions/872/what-is-the-cost-to-store-1kb-10kb-100kb-worth-of-data-into-the-ethereum-block`.

The cost of storing 1 KB is roughly 0.032 ETH. A decent image file is about 2 MB. 1 MB is 1,000 KB if you ask hard drive manufacturers, or 1,024 KB if you ask an operating system. We simply round this to 1,000 because it does not make any difference to our calculation. Consequently, the cost of storing a 2 MB file on Ethereum is around 2,000 multiplied by 0.032 ETH, which is equal to 64 ETH. The price of ETH is always changing. At the time of writing, the cost of 1 ETH is around 120 US dollars.

This means that to store a 2 MB picture file (a normal-sized stock picture file on Unsplash), you need to spend 7,680 US dollars. A one-and-a-half-minute video file in MP4 format is roughly 46 MB. Consequently, you need to spend 176,640 US dollars to store this video file on Ethereum.

Instead of paying this, blockchain developers will usually store the reference of a video file on the blockchain and store the video file itself on normal storage, such as on AWS. In a Vyper smart contract, you can use the `bytes` data type to do so:

```
cute_panda_video: bytes[128]
```

Then, you can store the link of the video that you store in AWS S3 (`https://aws.amazon.com/s3/`) in the smart contract:

```
cute_panda_video = "http://abucket.s3-website-us-west-2.amazonaws.com/
cute_panda_video.mp4"
```

This approach is all fine and dandy, but the problem is that you are dependent on AWS. If the company, Amazon, does not like your cute panda video, they could delete it, and the URL that is present in the smart contract becomes invalid. You could change the value of the `cute_panda_video` variable on the smart contract (unless you forbid it from doing so), but this situation causes inconveniences in our application. If you use the service from a centralized company, your faith is dependent on the whim of that company.

We can mitigate this problem by using decentralized storage, such as IPFS. Instead of a URL, we could store the IPFS path (or IPFS hash) as the value of the `cute_panda_video` variable, as shown in the following example:

```
cute_panda_video = "/ipfs/
QmWgMcTdPY9Rv7SCBusK1gWBRJcBi2MxNkC1yC6uvLYPwK"
```

Then, we can launch our IPFS daemon on AWS and other places, such as GCP. Consequently, if AWS censors our cute panda video, the IPFS path of our cute panda video is still valid. We could serve the video from other places, such as GCP. You could even host the video on the computer at your grandma's house. People who are addicted to cute panda videos could even pin the video and help us serve the cute panda video.

That's not all. Apart from hosting a cute panda video in a decentralized fashion, there are other values of the decentralized video-sharing application. This value relates to blockchain technology. Suppose we want to build a *like* (thumbs up) video feature. We could store the *like* value on the blockchain. This prevents corruption. Imagine that we want to build a voting contest for the cutest panda video with a prize of 10 BTC. If our contest application is created in a centralized fashion (using a table to keep the *like* value on a SQL database such as MySQL or PostgreSQL), we, as centralized admins, could hijack the winner using the following code:

```
UPDATE thumbs_up_table SET aggregate_voting_count = 1000000 WHERE
video_id = 234;
```

Of course, it is not this easy to cheat. You need to cover your tracks with database logs by ensuring that the aggregate counts match individual counts. This needs to be done subtly. Instead of adding a whopping number of votes, such as 1 million at once, you could add the aggregate count to a random number between 100 and 1,000 in an hour. This is not to suggest that you cheat your users – I am merely getting my point across.

With blockchain, we can prevent the integrity of data from becoming corrupted via the centralized admin. The *like* value is kept in the smart contract, and you let people audit the smart contract's source code. Our *like* feature on the decentralized video-sharing application increases the number of *likes* on a video through an honest process.

Other than the integrity of data, we could build a crypto economy. This means that we could have economic activities (such as selling, buying, bidding, and so on) in our smart contract. We could build tokens in the same smart contract. The coin of this token could be spent on *liking* the video so that liking videos is no longer free. The owner of the video could cash this out like money into their pocket. This dynamic could incentivize people to upload better videos.

On top of that, a decentralized application guarantees the independence of APIs. The nature of the decentralization of the application prevents APIs from being disturbed or harassed in a similar way to the Twitter API fiasco. A long time ago, developers could develop an interesting application on top of the Twitter API with a lot of freedom, but then Twitter imposed heavy restrictions on how developers could use their API. One such example is that Twitter once shut down API access to Politwoops, which preserved politicians' deleted tweets. Access has been reactivated, though. By making our application decentralized, we could increase the democratic nature of the API.

For educational purposes, our application has two main features. Firstly, you can see a list of videos, play videos, and upload videos. These are normal things that you do on YouTube. Secondly, you can *like* a video, but only with a coin or token.

Before we jump into building the application, let's design the architecture of the smart contract and the architecture of the web application.

The architecture of the video-sharing smart contract

Our application starts with the smart contract. There are a few things that our smart contract needs to do here:

- Keep track of videos that have been uploaded by a user.
- Utilize a token and its standard operation (ERC20).
- Provide a way for a user to like a video using a coin or token. The coin that's used for liking a video will be transferred to the video owner.

That's it. We always strive to keep the smart contract as short as possible. The more lines of code we have, the higher the chances of a bug showing up. *And a bug in a smart contract can't be fixed.*

Before writing the smart contract, let's think about how we want to structure it. The structure of the smart contract includes the data structure. Let's look at an example of a data structure we can use to track a user's videos.

We want to use a mapping variable with an address data type as the key. The difficult part here is choosing what data type we want to use as the value for this mapping data type. As we learned in *Chapter 3*, *Implementing Smart Contracts Using Vyper*, there is no infinite-size array in Vyper. If we use a `String` array, we are limited to a certain size of the array as the value for this mapping. This means a user can have a maximum amount of videos. We can use a `String` array to hold a list of videos that is very large, such as 1 million videos. What is the chance someone will upload more than 1 million videos? If you upload one video per day, you will only upload 3,650 videos in 10 years. The IPFS path – for example, `QmWgMcTdPY9Rv7SCBusK1gWBRJcBi2MxNkC1yC6uvLYPwK` – has a length of 44 characters. Consequently, you must use at least a `String[44]` data type, but we'll round this up to `String[50]`.

Instead, we want to have another mapping data type variable (let's call this *mapping z*) as the value of this mapping data type variable, as described in the previous paragraph. Mapping z has an integer as the key and a struct that contains a `String[50]` data type variable to keep the IPFS path and a `String[20]` data type variable to keep the video title as the value. There is an integer tracker to initiate the value of the key in mapping z. This integer tracker is initialized with a value of 0. Every time we add a video (IPFS path and video title) to mapping z, we increase this integer tracker by one. So, the next time we add another video, the key of mapping z is not 0 anymore, but 1. This integer tracker is unique for each account. We could create another mapping of the account for this integer tracker.

After taking care of the videos, we focus on the *likes*. How do we store the fact that user A likes video Z? We need to make sure that a user cannot *like* the same video more than once. The easiest way to do this is to create a mapping with a `String[100]` data type as the key and a `boolean` data type as the value. The `String[100]` data type variable is a combination of using the video liker's address, the video uploader's address, and the index of videos. The `boolean` data type variable is used to indicate whether the user has already liked the video or not.

On top of that, we need an integer data type to keep the aggregate count of the number of *likes* a video has. The aggregate likes is a mapping variable that is composed of a `String[100]` data type as the key and an integer data type as the value. The `String[100]` data type variable is a combination of the video uploader's address and the index of the videos.

The downside of this approach is that it is very hard to keep track of which users have liked particular videos in the smart contract. We could create another mapping to keep track of which users have liked a certain video. However, that would complicate our smart contract. Previously, we went the extra mile to create a mapping dedicated to keeping track of all videos that a user has uploaded. This is necessary because we want to get a list of a user's videos. This is what we call a core feature. However, keeping track of which users have liked a video is not what I call a core feature.

So long as we can make the video-liking process honest, we don't need to keep track of which users have liked a video. If we are itching to keep track of these users, we can use events in a smart contract. Every time a user *likes* a video, it triggers an event. Then, on the client side with the `ape` library, we could filter these events to get all users who like a particular video. This will be an expensive process and should be done separately from the main application. We can use background jobs using Celery, at which point the result can be stored on a database such as SQLite, PostgreSQL, or MySQL. Building a decentralized application does not mean completely negating a centralized approach.

> **Note**
> The topic of tokens will be discussed thoroughly in *Chapter 12*, *Creating Tokens on Ethereum*.

The architecture of the video-sharing web application

We will develop a Python web application that we'll use as the frontend for our smart contract. This means we need a proper server to become the host for the Python web application. For this, we at least need a Gunicorn web server. In other words, we need to host our Python web application in a centralized server, such as in AWS, GCP, or Azure. This is fine for viewing videos, but the problem arises when a user wants to upload a video because that requires accessing a private key. Users may become concerned that our Python web application since it's on a centralized server, would steal their private keys.

So, the solution is to post the source code of our Python web application on GitHub or GitLab, then tell a user to download it, install it, and run it on their computer. They can audit our Python web application's source code to make sure there is no pesky code trying to steal their private keys. However, if they need to audit the source code every time, then we must add another commit to our Git repository.

Or better still, we could store our Python web application's source code on IPFS. They can download this from IPFS and be sure that our application's source code cannot be tampered with. They only need to audit the source code once before using it.

However, while we could host a static website on IPFS, we can't do the same with dynamic web pages such as Python, PHP, Ruby, or Perl web applications. Such dynamic websites need a proper web server. Consequently, anyone who downloads our Python web application's source code needs to install the right software before executing our application. They need to install the Python interpreter, the web server (Gunicorn, Apache, or NGINX), and all of the necessary libraries.

However, only desktop users can do that. Mobile users cannot execute our application because there are no proper Python interpreters or web servers on the Android or iOS platforms.

This is where JavaScript shines. You could create a static and dynamic website so that you can have interactivity on your web pages. You could also create a complex JavaScript web application using React.js, Angular.js, Ember.js, or Vue.js and deploy it on IPFS. By doing this, both a desktop user and a mobile user could execute the JavaScript web application. Because this is a book about Python, we will still look at creating a Python web application. However, you should keep the advantages of JavaScript compared to Python in mind.

No matter how good JavaScript is, it still cannot save the plight of mobile users. *Computing power on mobile platforms is still less powerful than computing power on desktop.* You still cannot run a full Ethereum node on a mobile platform in the same way that you cannot run IPFS software on a mobile platform.

So, let's design our web application. It has a few utilities:

- Playing a video
- Uploading a video
- Liking a video
- Listing recent videos from many users

Playing a video involves accepting the video uploader's address and the index of the videos as parameters. If the video doesn't exist on our storage yet, we download it from IPFS. Then, we serve the video to the user.

Uploading a video requires interacting with an Ethereum node. This method or functionality to upload a video accepts an argument of the account's address to be used, an argument of a password for the encrypted private key, an argument of the video file, and an argument of the video title. First, we store the video file on IPFS. If this process succeeds, we can store information about this video on the blockchain.

Liking a video also requires interacting with an Ethereum node. This method or functionality to *like* a video accepts an argument of the video liker's address to be used, an argument of a password for the encrypted private key, an argument of the video uploader's address, and an argument of the video's index. After making sure that the user has not liked the video previously, we store this information on the blockchain.

Listing recent videos from many users is a bit tricky. The effort involved is quite tremendous. In a smart contract, we don't have a variable to track all participating users. We also don't have a variable to track all videos from different users. However, we can create an event through the method of storing video information on the blockchain. After doing so, we can find all recent videos from this event.

Now, it's time to build our decentralized video-sharing application.

Writing the video-sharing smart contract

Without further ado, let's set up our smart contract development platform.

First things first, we must set up our virtual environment, as follows:

```
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install --upgrade pip
(.venv) $ pip install eth-ape'[recommended-plugins]'==0.7.23
```

Then, we must create a directory and initialize it as the project directory:

```
(.venv) $ mkdir videos_sharing_smart_contract
(.venv) $ cd videos_sharing_smart_contract
(.venv) $ ape init
Please enter project name: VideoSharing
SUCCESS: VideoSharing is written in ape-config.yaml
```

Now, we must write our smart contract code in `videos_sharing_smart_contract/contracts/VideoSharing.vy`, as shown in the following code block (refer to the code file at the following GitHub link for the full code: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_10/videos_sharing_smart_contract/contracts/VideoSharing.vy`):

```
# @version ^0.3.0

struct Video:
    path: String[50]
    title: String[20]

event Transfer:
    _from: indexed(address)
    _to: indexed(address)
    _value: uint256

event Approval:
    _owner: indexed(address)
    _spender: indexed(address)
    _value: uint256
...
@external
@view
def video_aggregate_likes(_user_video: address, _index: uint256) ->
uint256:
```

```
    _user_video_str: bytes32 = convert(_user_video, bytes32)
    _index_str: bytes32 = convert(_index, bytes32)
    _key: Bytes[100] = concat(_user_video_str, _index_str)
    return self.aggregate_likes[_key]
```

Now, let's discuss our smart contract bit by bit:

```
# @version ^0.3.0

struct Video:
    path: String[50]
    title: String[20]
```

This is a struct of the video information that we want to keep on the blockchain. The `path` variable of the `Video` struct stores the IPFS path, which has a length of 44. The IPFS path will be a different length if we use another hashing function. Remember that IPFS uses multihash when hashing objects. If you use a more expensive hashing function, such as SHA512, in your IPFS configuration, then you need to double the size of the `String` data type. For example, `String[100]` should be sufficient. The `title` variable of the `Video` struct stores the video title. Here, I used `String[20]` because I wanted to keep the title short. You could use a lengthier `String`, such as `String[100]`, if you want to store a lengthier title. However, remember that the more bytes you store on the blockchain, the more gas (money!) you have to spend. Of course, you could add more information in this struct, such as a video description or video tags, so long as you know the consequences, which is more gas needed to execute the method to store the video information.

Now, let's consider the list of events, as shown in the following code:

```
event Transfer:
    _from: indexed(address)
    _to: indexed(address)
    _value: uint256

event Approval:
    _owner: indexed(address)
    _spender: indexed(address)
    _value: uint256

event UploadVideo:
    _user: indexed(address)
    _index: uint256

event LikeVideo:
    _video_liker: indexed(address)
    _video_uploader: indexed(address)
    _index: uint256
```

Here, `Transfer` and `Approval` are part of ERC20 standard events. You'll learn more about ERC20 in *Chapter 12*, *Creating Tokens on Ethereum*. The `UploadVideo` event is triggered when we upload video information in our smart contract. We save the video uploader's address and the index of videos. The `LikeVideo` event is triggered when we like a video in our smart contract.

Now, we must save the video liker's address, the video uploader's address, and the index of videos:

```
user_videos_index: HashMap[address, uint256]
```

This is the integer tracker for our unlimited array. So, if `user_videos_index[address of user A] = 5`, this means user A has uploaded four videos already.

The following is part of the ERC20 standard:

```
name: public(String[20])
symbol: public(String[3])
totalSupply: public(uint256)
decimals: public(uint256)
balances: HashMap[address, uint256]
allowed: HashMap[address, HashMap[address, uint256]]
```

Refer to *Chapter 12*, *Creating Tokens in Ethereum*, for more information about ERC20.

Let's move on to the next line:

```
all_videos: HashMap[address, HashMap[uint256, Video]]
```

This is the core variable that will keep all videos from all users. The `address` data type key is used to hold a user's address. The `HashMap[uint256, Video]` data type value is our infinite array. The `uint256` key in `HashMap[uint256, Video]` starts from 0 and is then tracked by the `user_videos_index` variable. Finally, the `Video` struct is our video information.

The next two lines of code are used for the *likes*:

```
likes_videos: HashMap[Bytes[100], bool]
aggregate_likes: HashMap[Bytes[100], uint256]
```

The `likes_videos` variable is used to check whether a certain user has liked a particular video or not. The `aggregate_likes` variable is used to show how many likes this particular video already has.

Now that we've defined our variables, let's consider the following code block:

```
@external
def __init__():
    _initialSupply: uint256 = 500
    _decimals: uint256 = 3
    self.totalSupply = _initialSupply * 10 ** _decimals
```

```
    self.balances[msg.sender] = self.totalSupply
    self.name = 'Video Sharing Coin'
    self.symbol = 'VID'
    self.decimals = _decimals
    log Transfer(ZERO_ADDRESS, msg.sender, self.totalSupply)
...
@external
@view
def allowance(_owner: address, _spender: address) -> uint256:
    return self.allowed[_owner][_spender]
```

These are the standard ERC20 functions that you'll learn about in *Chapter 12*.

Now, let's look at the function that uploads the video:

```
@external
def upload_video(_video_path: String[50], _video_title: String[20]) ->
bool:
    _index: uint256 = self.user_videos_index[msg.sender]
    self.all_videos[msg.sender][_index] = Video({ path: _video_path,
title: _video_title })
    self.user_videos_index[msg.sender] += 1
    log UploadVideo(msg.sender, _index)
    return True
```

This method is used to store video information on the blockchain. We call this method after we upload the video to IPFS. Here, _video_path is the IPFS path, and _video_title is the video title. We get the latest index from the video uploader (msg.sender). Then, we set the value of the Video struct to all_videos based on the video uploader's address and the latest index. Finally, we increase the integer tracker (`user_videos_index`). Don't forget to log this event.

Now, we need to add convenience functions to get the latest video index, the video IPFS path, and the video title for clients, as shown in the following code:

```
@external
@view
def latest_videos_index(_user: address) -> uint256:
    return self.user_videos_index[_user]

@external
@view
def videos_path(_user: address, _index: uint256) -> String[50]:
    return self.all_videos[_user][_index].path

@external
```

```
@view
def videos_title(_user: address, _index: uint256) -> String[20]:
    return self.all_videos[_user][_index].title
```

Without these methods, you could still get information about the video, but accessing a struct variable inside a nested mapping data type variable is not straightforward.

The following code shows the method that's used to *like* a video. It accepts the video uploader's address and the video's index. Here, you create two keys – one for `likes_videos` and the other for `aggregate_likes`. The key for `likes_videos` is a combination of the video liker's address, the video uploader's address, and the video's index. The key for `aggregate_likes` is a combination of the video uploader's address and the video's index. After creating keys, we must make sure the video liker cannot like the same video in the future and that the video liker has not liked this particular video before. Liking a video merely sets a `True` value for the `likes_videos` variable with the key that we have created. Then, we increase the value of `aggregate_likes` with the key that we have created by `1`. Finally, we transfer one coin of the token from the video liker to the video uploader. Don't forget to log this event, as shown in the following code:

```
@external
def like_video(_user: address, _index: uint256) -> bool:
    _msg_sender_str: bytes32 = convert(msg.sender, bytes32)
    _user_str: bytes32 = convert(_user, bytes32)
    _index_str: bytes32 = convert(_index, bytes32)
    _key: Bytes[100] = concat(_msg_sender_str, _user_str, _index_str)
    _likes_key: Bytes[100] = concat(_user_str, _index_str)
    assert _index < self.user_videos_index[_user]
    assert self.likes_videos[_key] == False
    self.likes_videos[_key] = True
    self.aggregate_likes[_likes_key] += 1
    self._transfer(msg.sender, _user, 1)
    log LikeVideo(msg.sender, _user, _index)
    return True
```

The following lines of code are convenience methods that are used to check whether a video has been liked by a particular user and how many likes this particular video has already:

```
@external
@view
def video_has_been_liked(_user_like: address, _user_video: address,
_index: uint256) -> bool:
    _user_like_str: bytes32 = convert(_user_like, bytes32)
    _user_video_str: bytes32 = convert(_user_video, bytes32)
    _index_str: bytes32 = convert(_index, bytes32)
    _key: Bytes[100] = concat(_user_like_str, _user_video_str, _index_
str)
```

```
    return self.likes_videos[_key]

@external
@view
def video_aggregate_likes(_user_video: address, _index: uint256) ->
uint256:
    _user_video_str: bytes32 = convert(_user_video, bytes32)
    _index_str: bytes32 = convert(_index, bytes32)
    _key: Bytes[100] = concat(_user_video_str, _index_str)
    return self.aggregate_likes[_key]
```

To compile the smart contract, you can run the following command inside the `videos_sharing_smart_contract` directory:

```
(.venv) $ ape compile
```

Once the smart contract is ready, we can run the private blockchain with `geth`:

```
$ geth -dev -http -http.api eth,web3,net
```

For this chapter, you need at least two Ethereum accounts. You can reuse the accounts that you created in previous chapters, but you can also create an account with the following command:

```
(.venv) $ ape accounts generate video
```

The preceding command will create an Ethereum account named `video`. The file is located at `~/.ape/accounts/video.json`.

To connect to the private blockchain, run the following command in a separate terminal/console:

```
$ geth attach /tmp/geth.ipc
```

Then, send some ETH to at least two accounts from the `first` account by running the following command inside the `geth` console:

```
> eth.sendTransaction({from:  eth.accounts[0], to:
"0x05b480862bf5875a1d5dd277b67cc6e687053413", value: web3.toWei(50,
"ether")})
```

Change `0x05b480862bf5875a1d5dd277b67cc6e687053413` to your account address. Run this command again but for your second account address.

Now, let's write a deployment script. Create a file named `deploy.py` inside the `scripts` folder and add the following code to it:

```
from ape import accounts, project
import os


def main():
    password = os.environ["VIDEO_ACCOUNT_PASSWORD"]
    account = os.environ["VIDEO_ACCOUNT"]
    deployer = accounts.load(account)
    deployer.set_autosign(True, passphrase=password)
    contract = project.VideoSharing.deploy(sender=deployer)
    print(f"The contract address is {contract.address}")
```

This script deploys the smart contract to the blockchain and displays the contract's address.

To run the script, you must set the necessary environment variables:

```
$ export VIDEO_ACCOUNT_PASSWORD=yourpassword
$ export VIDEO_ACCOUNT=youraccount
```

Then, you can run the script like so:

```
(.venv) $ ape run deploy --network ethereum:local:geth
INFO: Connecting to existing Geth node at http://localhost:8545/
[hidden].
WARNING: Danger! This account will now sign any transaction it's
given.
WARNING: Using cached key for video
INFO: Confirmed
0x976b8e5d08a09805db06b78cb983b2a22752c0c3b83d40bfbdef3c0f7af31658
(total fees paid = 429868336539936)
SUCCESS: Contract 'VideoSharing' deployed to:
0xb3f42c359A93225e4212b2fABdb61Af76bAE9C81
The contract address is 0xb3f42c359A93225e4212b2fABdb61Af76bAE9C81
```

Ensure you keep the address. Now that the smart contract has been deployed to the blockchain, let's prepare the accounts. In the smart contract, an account needs the ERC20 token to like a video. So, you must send the token from the deployer account to another account.

Creating a Bootstrap script

In the smart contract, the deployer has all the tokens. So, it has to distribute them either through sales or auctions. For our purpose, let's be generous and transfer the tokens to other addresses for free.

Create a new file inside the `scripts` folder and name it `bootstrap.py`. Add the following code to it:

```
from ape import accounts, project
import os

def main():
    address = os.environ["VIDEO_SHARING_ADDRESS"]
    password = os.environ["VIDEO_ACCOUNT_PASSWORD"]
    destination = os.environ["DESTINATION_ADDRESS"]
    account = os.environ["VIDEO_ACCOUNT"]
    deployer = accounts.load(account)
    deployer.set_autosign(True, passphrase=password)
    contract = project.VideoSharing.at(address)
    contract.transfer(destination, 100, sender=deployer)
```

In the preceding script, you get the `VideoSharing` smart contract reference from the blockchain. Then, you execute the `transfer` method of `VideoSharing` with the recipient as the first argument and the number of tokens to be transferred as the second argument.

Next, you set the destination address using the `DESTINATION_ADDRESS` environment variable.

Before executing the script, you need to set up the necessary environment variables:

```
(.venv) $ export DESTINATION_ADDRESS=destinationaddress
(.venv) $ export VIDEO_SHARING_ADDRESS= yoursmartcontractaddress
(.venv) $ export VIDEO_ACCOUNT_PASSWORD=youraccountpassword
(.venv) $ export VIDEO_ACCOUNT=youraccountname
```

Then, you can run the Bootstrap script, like so:

```
(.venv) $ ape run bootstrap --network ethereum:local:geth
```

After executing the script, the destination address has 100 VID tokens. If you look at the smart contract, the token is named `Video Sharing Coin` with VID as the symbol. This is the constructor method of the smart contract you've written:

```
@external
def __init__():
    _initialSupply: uint256 = 500
    _decimals: uint256 = 3
    self.totalSupply = _initialSupply * 10 ** _decimals
    self.balances[msg.sender] = self.totalSupply
    self.name = 'Video Sharing Coin'
    self.symbol = 'VID'
    self.decimals = _decimals
    log Transfer(ZERO_ADDRESS, msg.sender, self.totalSupply)
```

You can execute the script again with different destination addresses. You just need to change the value of the `DESTINATION_ADDRESS` environment variable before executing the script.

Then, you need to upload some videos to decentralized storage so that the application doesn't look so empty when you launch the application later.

Create a directory named `stock_videos` adjacent to the `videos_sharing_smart_contract` folder and download videos to this folder. For this chapter, you can download videos from `https://www.pexels.com/videos/`. You can place 6 to 10 videos as a starting point.

Next, create a script that will upload these videos to IPFS inside the `scripts` folder and name it `upload.py`. Add the following code to it:

```
from ape import accounts, project
import aioipfs
import asyncio
import os

async def add(file_path):
    client = aioipfs.AsyncIPFS()

    files = [file_path]
    hash = None
    async for added_file in client.add(files):
        hash = added_file['Hash']
    await client.close()
    return hash

def main():

    address = os.environ["VIDEO_SHARING_ADDRESS"]
    password = os.environ["VIDEO_ACCOUNT_PASSWORD"]
    account = os.environ["VIDEO_ACCOUNT"]
    deployer = accounts.load(account)
    deployer.set_autosign(True, passphrase=password)
    contract = project.VideoSharing.at(address)

    directory = '../stock_videos'
    movies = os.listdir(directory)
    for index, movie in enumerate(movies):
        movie_path = directory + '/' + movie
        loop = asyncio.get_event_loop()
        ipfs_path = loop.run_until_complete(add(movie_path))
        title = movie.rstrip('.mp4')[:19]
        contract.upload_video(ipfs_path, title, sender=deployer)
```

In the preceding script, you get all movie files from the `stock_videos` directory, which is located in the parent directory of the `scripts` directory. Then, you add each move file with the `add` method of the IPFS connection object to IPFS. After uploading the movie file, you get the IPFS path. Then, you get the title from the filename of the video file. Finally, you upload this information to the smart contract with the `upload_video` method.

In a nutshell, you upload the content of the video to IPFS, but you upload the path of the video in IPFS to the smart contract.

Let's execute the uploading videos script. Make sure the IPFS daemon is running. You can run the script with the `ape` command:

```
(.venv) $ ape run upload --network ethereum:local:geth
```

With that, your videos have been stored in IPFS and you also stored the information in the smart contract. Now, it's time to build the web application.

Building the video-sharing web application

Previously, we created a desktop application using the PySide library. This time, we're going to build a web application using the Django library. We're using the same Python virtual environment that we used to write the smart contract. So, make sure the `aioipfs` library is installed before going through the following steps:

1. Without further ado, let's install Django:

   ```
   (.venv) $ pip install Django
   ```

2. We also need the OpenCV Python library to get the thumbnails of our videos:

   ```
   (.venv) $ pip install opencv-python
   ```

3. Now, let's create our Django project directory. This will create a skeleton Django project, along with its settings files:

   ```
   (.venv) $ django-admin startproject decentralized_videos
   Inside this new directory, create the static directory and the
   media directory:
   (.venv) $ cd decentralized_videos
   (.venv) $ mkdir static media
   ```

4. In the same directory, create a Django application named `videos`:

   ```
   (.venv) $ python manage.py startapp videos
   ```

5. Then, update our Django project settings file. This file is located in `decentralized_videos/settings.py`. Add our new application, `videos`, to the `INSTALLED_APPS` variable. Make sure there is a comma between the `videos` and `django.contrib.staticfiles` strings. We need to add every Django application to this variable for the Django project to recognize it. A Django project can be composed of many Django applications, as shown in the following code:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'videos'
]
```

Then, in the same file, add the following lines of code:

```
STATIC_URL = 'static/'
STATICFILES_DIRS = [
    Path(BASE_DIR, "static"),
]
MEDIA_URL = 'media/'
MEDIA_ROOT = Path(BASE_DIR, 'media')
```

The `STATIC_URL` variable defines how we access a static URL. With this value, we can access static files via `http://localhost:8000/static/our_static_file`. The `STATICFILES_DIRS` variable refers to where we keep our static files in the filesystem. We simply store the videos in the static directory inside our Django project directory. Here, `MEDIA_URL` has the same purpose as `STATIC_URL` but for media files. Media files are what users upload into the Django project, while static files are what we, as developers, put into the Django project.

Views

Now, let's create the view file of the video application. *A view is a controller that is like an API endpoint*. The file is located in `decentralized_videos/videos/views.py`. Refer to the code file at the following GitHub link for the full code: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_10/decentralized_videos/videos/views.py`.

A view is a controller or a gateway. This is where you receive requests from users. Now, let's consider models, which is where you interact with the blockchain.

Models

Let's create our models file in `decentralized_videos/videos/models.py`. Most logic and heavy operations happen here. Calling a smart contract's methods and storing files in IPFS also happen here. Refer to the code file at the following GitHub link for the full code: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_10/decentralized_videos/videos/models.py`.

Let's discuss the core functionalities of our Django project bit by bit. First, we import convenience methods from the Python standard library, the IPFS Python library, the OpenCV Python library, and the Ape library, as well as, some variables from the Django settings file, as shown in the following code:

```
import os.path, json
import asyncio
import aioipfs
import cv2
import os
```

After the import lines, there is omitted code because the code is too long. Now, you must import some variables from the settings file and declare some constants, as shown here:

```
from decentralized_videos.settings import STATICFILES_DIRS, STATIC_
URL, BASE_DIR, MEDIA_ROOT

BLOCKCHAIN_NETWORK = "local"
BLOCKCHAIN_PROVIDER = "geth"
```

Then, we must set up the async functions to access IPFS:

```
loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)

async def get(ipfs_path, download_path):
    client = aioipfs.AsyncIPFS()

    await client.get(ipfs_path, dstdir=download_path)
    await client.close()

async def add(file_path):
    client = aioipfs.AsyncIPFS()

    files = [file_path]
    hash = None
    async for added_file in client.add(files):
        hash = added_file['Hash']
```

```
    await client.close()
    return hash
```

Now, we can start initializing the code of the `VideosSharing` model:

```
class VideosSharing:

    def __init__(self):
        self.address = Web3.to_checksum_address(os.environ["VIDEO_
SHARING_ADDRESS"])
        with open('../videos_sharing_smart_contract/.build/
VideoSharing.json') as f:
            contract = json.load(f)
            self.abi = contract['abi']
```

We initialize this instance by creating an object of the ABI of the smart contract.

Then, we have a method that's used in the `index` view, called `recent_videos`:

```
    def recent_videos(self, amount=20):
        with networks.ethereum[BLOCKCHAIN_NETWORK].use_
provider(BLOCKCHAIN_PROVIDER):
            ct = ContractType.parse_obj({"abi": self.abi})
            self.SmartContract = Contract(self.address, ct)
            events = self.SmartContract.UploadVideo.query("*", start_
block=0)

        videos = []
        for event in events["event_arguments"]:
            video = {}
            video['user'] = event['_user']
            video['index'] = event['_index']
            video['path'] = self.get_video_path(video['user'],
video['index'])
            video['title'] = self.get_video_title(video['user'],
video['index'])
            video['thumbnail'] = self.get_video_
thumbnail(video['path'])
            videos.append(video)
        videos.reverse()
        return videos[:amount]
```

To gain access to the blockchain, you can use the provider alongside the `use_provider` method and a `with` statement, as shown in the following code:

```
with networks.ethereum[BLOCKCHAIN_NETWORK].use_provider(BLOCKCHAIN_
PROVIDER):
```

Recent videos can be retrieved from events. As you may recall when we uploaded a video in our smart contract, we logged an event here. Our event is `UploadVideo`. Because this Django project is a toy application, we get all the events from the starting block. In the real world, you will want to limit it (maybe the last 100 blocks). Furthermore, you probably want to store events in a database in background jobs (such as cron) for easy retrieval. This event object contains the video uploader and the video's index. Based on this information, we can get the video path, the video title, and the video thumbnail. We accumulate videos in the `videos` object, reverse it (because we want to get recent videos), and return this object to the caller of the method.

The `get_video_path` and `get_video_title` methods are used to get the video path and the video title based on the address of the video uploader, respectively.

Here, we use the `videos_path` and `videos_title` methods from our smart contract.

The following code block contains the method that gets the video thumbnail:

```
    def get_video_thumbnail(self, ipfs_path):
        thumbnail_file = str(STATICFILES_DIRS[0]) + '/' + ipfs_path +
'.png'
        url_file = STATIC_URL + '/' + ipfs_path + '.png'

        if os.path.isfile(thumbnail_file):
            return url_file
        else:
            return "https://bulma.io/images/placeholders/640x480.png"
```

When we view a video on the video-playing page, we can check whether there is a certain filename with a `.png` file extension. We can find this filename pattern inside the static files directory. If we can't find the file, we can just use a placeholder picture file from the internet.

The following code block contains the method for retrieving a specific video:

```
    def get_video(self, user, index):
...
        with networks.ethereum[BLOCKCHAIN_NETWORK].use_
provider(BLOCKCHAIN_PROVIDER):
            ct = ContractType.parse_obj({"abi": self.abi})
            self.SmartContract = Contract(self.address, ct)
            video['aggregate_likes'] = self.SmartContract.video_
aggregate_likes(user, index)
```

```
        if os.path.isfile(video_file):
            video['url'] = STATIC_URL + '/' + ipfs_path + '.mp4'
        else:
            loop.run_until_complete(get(ipfs_path, str(BASE_DIR)))
            video['url'] = STATIC_URL + '/' + ipfs_path + '.mp4'
            os.rename(str(BASE_DIR) + '/' + ipfs_path,
str(STATICFILES_DIRS[0]) + '/' + ipfs_path + '.mp4')

        if not os.path.isfile(thumbnail_file):
            self.process_thumbnail(ipfs_path)

        return video
```

This is used in the `video` view. Here, we need the video path, the video title, the video file, the video thumbnail, and the aggregate likes of this video (which we can get with the `video_aggregate_likes` method from our smart contract). We must check whether this MP4 file exists or not in our static files directory. If not, we retrieve it from IPFS with the `client.get` method. Then, we move the file to the static files directory and create a thumbnail image if one does not exist yet.

The following method demonstrates what happens when we upload a video:

```
    def upload_video(self, video_user, password, video_file, title):
        video_path = str(MEDIA_ROOT) + '/video.mp4'
        with open(video_path, 'wb+') as destination:
            for chunk in video_file.chunks():
                destination.write(chunk)
        ipfs_path = loop.run_until_complete(add(video_path))
        title = title[:19]
        with networks.ethereum[BLOCKCHAIN_NETWORK].use_
provider(BLOCKCHAIN_PROVIDER):
            ct = ContractType.parse_obj({"abi": self.abi})
            self.SmartContract = Contract(self.address, ct)
            sender = accounts.load(video_user)
            sender.set_autosign(True, passphrase=password)
            self.SmartContract.upload_video(ipfs_path, title,
sender=sender)
```

We save the file in the media directory from our file in memory and then add the file to IPFS with the `client.add` method. We get the IPFS path and prepare the title of the video. Then, we call the `upload_video` method from our smart contract. Remember to set enough gas and a gas price for this. This is quite an expensive smart contract method. We wait for the transaction to be confirmed. In the real world, you'll want to do all of these steps using a background job.

The `process_thumbnail` method shows how to generate a thumbnail from a video:

```
    def process_thumbnail(self, ipfs_path):
        thumbnail_file = str(STATICFILES_DIRS[0]) + '/' + ipfs_path +
'.png'
        if not os.path.isfile(thumbnail_file):
            video_path = str(STATICFILES_DIRS[0]) + '/' + ipfs_path +
'.mp4'
            cap = cv2.VideoCapture(video_path)
            cap.set(cv2.CAP_PROP_POS_FRAMES, 0)
            _, frame = cap.read()
            cv2.imwrite(thumbnail_file, frame)
```

After ensuring no such file exists, we get the video object. We read the first frame of the object and save this to an image file. This video functionality is from the OpenCV Python library.

Then, we have the method for liking a video:

```
    def like_video(self, video_liker, password, video_user, index):
        with networks.ethereum[BLOCKCHAIN_NETWORK].use_
provider(BLOCKCHAIN_PROVIDER):
            ct = ContractType.parse_obj({"abi": self.abi})
            self.SmartContract = Contract(self.address, ct)
            sender = accounts.load(video_liker)
            if self.SmartContract.video_has_been_liked(sender.address,
video_user, index):
                return
            sender.set_autosign(True, passphrase=password)
            self.SmartContract.like_video(video_user, index,
sender=sender)
```

We make sure this video has not been liked by calling the `video_has_been_liked` method from our smart contract. Then, we call the `like_video` method with the required parameters from our smart contract.

Finally, we create an instance of the `VideosSharing` class so that we can import this instance:

```
videos_sharing = VideosSharing()
```

Instead of importing a class, I prefer to import an instance of a class. Consequently, we initialize a class instance here.

Next, we'll move on to templates or the HTML files.

Templates

It's time to write our templates. First, let's create a template directory using the following command lines:

```
(.venv) $ cd decentralized_videos
(.venv) $ mkdir -p videos/templates/videos
```

Then, we must create our base layout using the following lines of HTML. This is the layout that will be used by all our templates. The file is located at `videos/templates/videos/base.html`. You can refer to the following GitHub link for the full code: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_10/decentralized_videos/videos/templates/videos/base.html`.

Now, let's create our first template file in `videos/templates/videos/index.html`. Use the following code block to create the template file:

```
{% extends "videos/base.html" %}

{% block content %}
<section class="section">
  <div class="container">
    {% for video in videos %}
      {% cycle '<div class="columns">' '' '' '' %}
        <div class="column">
          <div class="card">
            <div class="card-image">
              <figure class="image is-4by3">
                <img src="{{ video.thumbnail }}" />
              </figure>
            </div>
            <p class="card-footer-item">
              <span><a href="{% url 'video' video_user=video.user
index=video.index %}">{{ video.title }}</a></span>
            </p>
          </div>
        </div>
      {% cycle '' '' '' '</div>' %}
    {% endfor %}
  </div>
</section>
{% endblock %}
```

Now, we will create our template file to play the video in `videos/templates/videos/video.html`. You can refer to the following GitHub link for the full code: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_10/decentralized_videos/videos/templates/videos/video.html`.

Finally, let's create the last template file for uploading videos in `videos/templates/videos/upload.html`. You can refer to the following GitHub link for the full code: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_10/decentralized_videos/videos/templates/videos/upload.html`.

In the next section, we'll learn how to route URLs to views.

URLs

The `urls` file is a routing mechanism in Django. Open `decentralized_videos/videos/urls.py`, delete its content, and replace it with the following script:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name='index'),
    path('video/<str:video_user>/<int:index>', views.video,
name='video'),
    path('upload-video', views.upload, name='upload'),
    path('like-video', views.like, name='like'),
]
```

Remember the views file we created previously? Here, we're mapping the views into routing. We can access the video-playing page by going to `http://localhost:8000/video/0x0000000000000000000000000000000000000000/1`. The parameters will be mapped to the `video_user` variable and the `index` variable. The first parameter of the `path` method is the way we call it in the browser. The second method is the view we use, and the third parameter is the name of the routing that is used in templates.

Now, we need to register these URLs to the project's `urls` file. Edit `decentralized_videos/decentralized_videos/urls.py` and add our `videos.urls` path so that our web application knows how to route our URL to our video views:

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
```

```
    path('', include('videos.urls')),
    path('admin/', admin.site.urls),
]
```

With this change, visitors' requests will be redirected to the proper places in the backend of the website.

Demo

Now, it's time for you to enjoy the fruits of your labor. Make sure you are inside the `decentralized_videos` directory before running the server. Don't forget to run the private blockchain and IPFS daemon first:

```
(.venv) $ cd decentralized_videos
(.venv) $ python manage.py runserver
```

Then, open `http://localhost:8000`. Here, you will be greeted with recent videos, as shown in the following screenshot. If you are confused about why I have a thumbnail for some videos, you need to go to the video-playing page to generate the thumbnail:

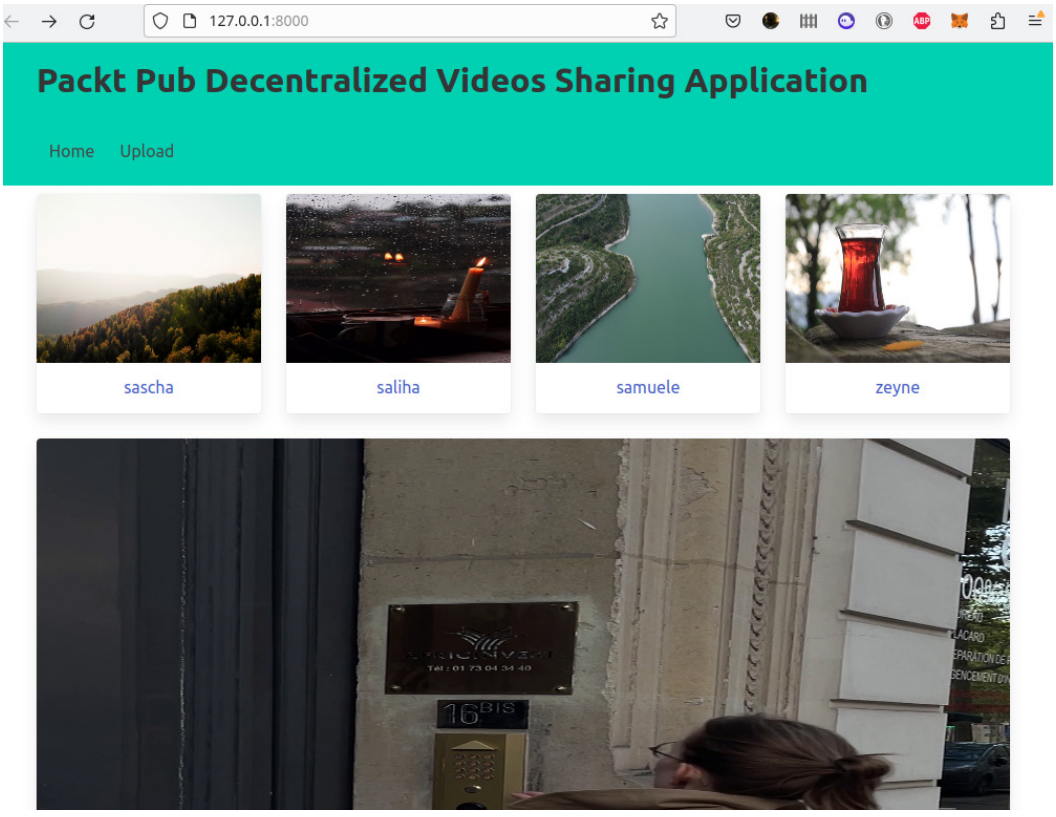


Figure 10.2: The index page

Let's click on one of the videos:

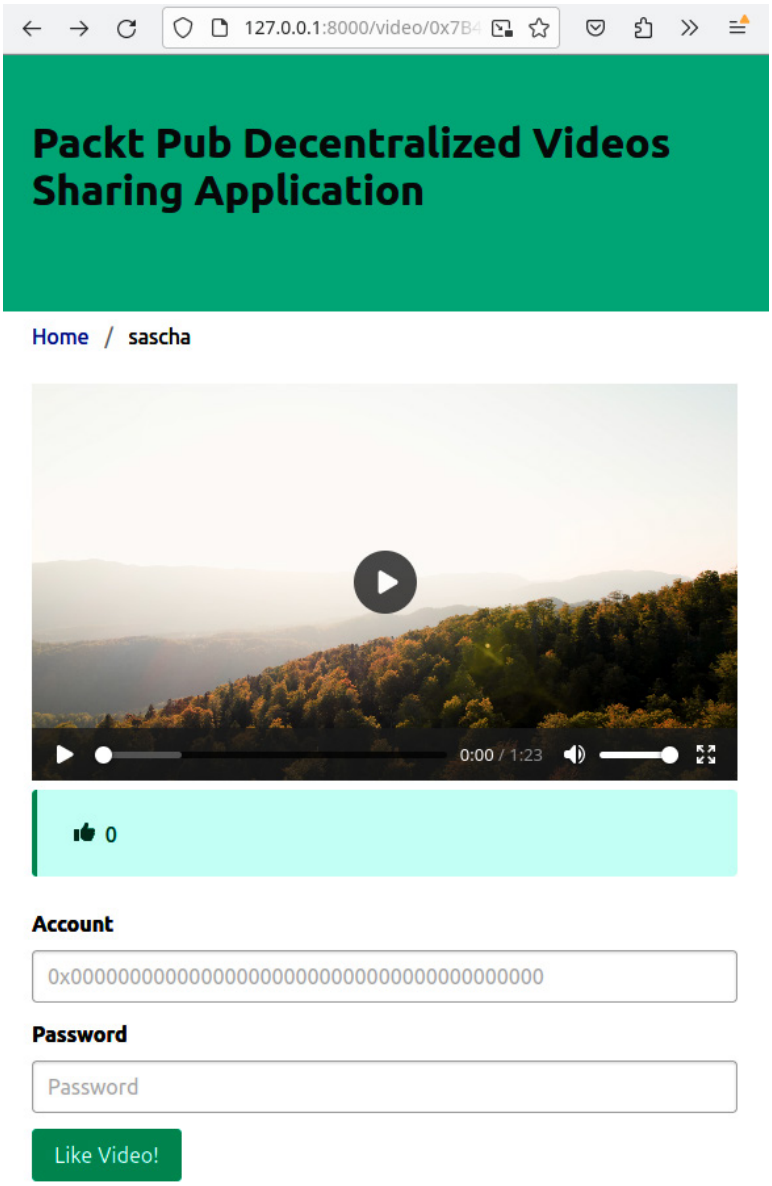


Figure 10.3: The video page

This is where you can play videos. You can also like the video in the form.

Finally, let's go to the uploading video page. Click the **Upload** link in the navigation menu:

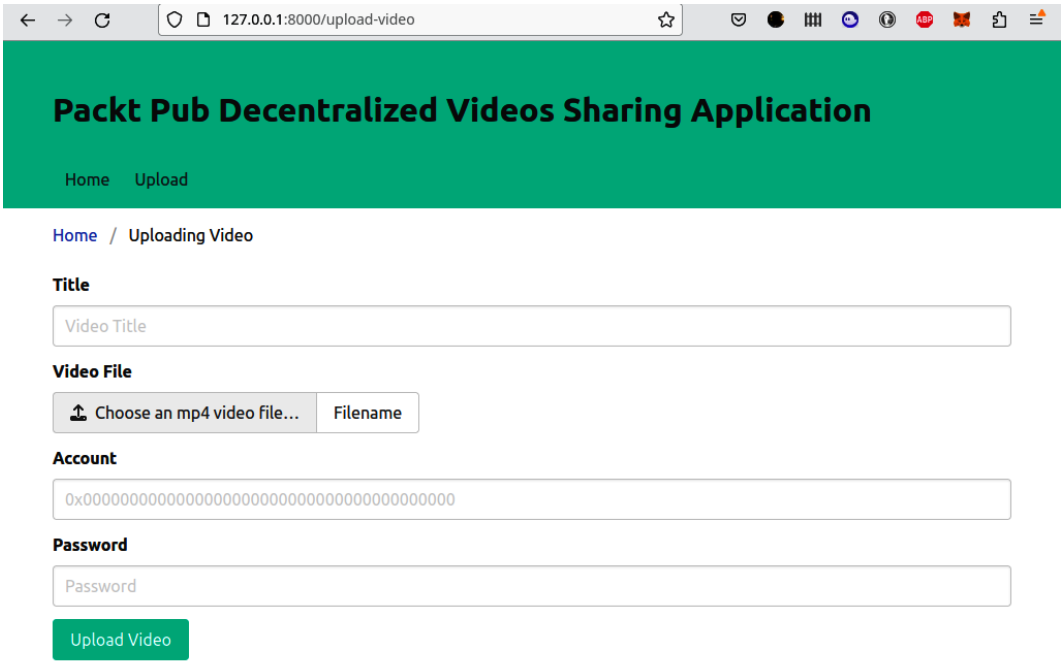


Figure 10.4: The uploading page

This is where you can upload a video.

Summary

In this chapter, we combined IPFS technology and smart contract technology to build a decentralized video-sharing application. First, we wrote a smart contract to store video information and video titles. We also built in crypto economics by making the act of liking videos require coins from the ERC20 token. In addition to this, we learned that even storing video information such as a string of the IPFS path and the title requires more gas than usual. After writing a smart contract, we built a web application using the Django library. We created a project, followed by building an application inside this project. Moving forward, we built views, models, templates, and URLs. In the models, we stored the video file in IPFS and then stored the IPFS path on the blockchain. We made the templates more beautiful using the Bulma CSS framework and then launched the application by executing the functionalities of this web application. This chapter will certainly help us in scaling data from blockchain applications with storage. In the next chapter, we will scale values and transactions from blockchain applications with **Layer Two** (**L2**) technology.

11
Exploring Layer 2

In this chapter, we learn what **Layer 2** (**L2**) is, what drives this technology, what are examples of L2, and how to use this technology. We'll see the limitations of the Ethereum blockchain and what L2 brings to the table to mitigate Ethereum's limitations. We also learn the bridge technology that is related to L2. This technology is crucial for connectivity between L2 and Ethereum.

The following topics will be covered in this chapter:

- Understanding L2
- Looking at examples of L2
- Deploying smart contracts to L2
- Introducing the bridge technology

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_11`.

Understanding L2

Before you understand what L2 is, it's best to understand the motivation behind it.

The Ethereum blockchain network is not the only blockchain. There are other blockchain networks, such as **BNB Smart Chain** (**BSC**), Solana, and Tron. There are a lot of users on these networks. One of the things that attracts users from the Ethereum network to other networks is lower transaction fees. Creating transactions on Ethereum can cost $5-$100 depending on what kind of transactions you create (sending ETH, deploying a smart contract, swapping tokens) and whether there are a lot of transactions at the time you create transactions. Compare this to Solana where the transaction fee is less than one cent of the US Dollar.

Also, the maximum number of transactions that Ethereum can handle right now is around 20-30 transactions per second. This is still low compared to the amount of ride transactions Uber can handle in the US, which is 266 rides. Even if you compare it to another blockchain such as Solana, Ethereum's performance is low. Solana can handle 65,000 transactions per second.

So, Ethereum has some limitations that are not scalable (compared to a popular application such as Uber in the US and another blockchain such as Solana).

If this is the case, why don't other blockchains win over all users from Ethereum? Because Ethereum has other valuable properties, which are security and decentralization.

To achieve speed and low transaction fees, other blockchain networks sacrifice other things such as decentralization or security. In Ethereum, there are a lot of nodes that become gatekeepers of which transactions can get recorded on the blockchain. If a node is trying to cheat, other nodes can check whether this cheating node is honest or not. Then, if proven to be cheating, other nodes can give a penalty to this node. This process is complex, and it's one of the reasons why putting transactions on Ethereum is quite expensive.

But imagine if you are the owner of all the nodes in Ethereum; you can simplify the process. There will be no cheating nodes because why would you cheat yourself? Remember that you own all the nodes. So, you can make the transaction fees lower. But the downside of this situation is the Ethereum network is not decentralized anymore.

That's why Ethereum still has many users. It's the second most valued cryptocurrency behind Bitcoin. Yes, the transaction fee is expensive, but the security and decentralized aspects are also important.

Still, people want to have their cake and eat it too. Can users have all of the good things (security, scalability, decentralization)?

That's where L2 technology comes in. L2 is a separate blockchain like Solana or BNC, but it piggybacks on Ethereum. So, it's still related to Ethereum. But as an end user, you can treat L2 as a totally different blockchain, just like Solana or BNC. It's scalable. The transaction fee is low. Security is guaranteed by Ethereum.

How L2 works

So, how does it work? There are different kinds of L2 on top of Ethereum. They have different mechanisms for using Ethereum as their source of security. But to get the point across, you can think of the way L2 works.

Imagine many transactions are happening on top of this L2 blockchain. You send 2 ETH to your friend. Your neighbor accesses a smart contract method; for example, they might vote on a voting smart contract deployed on L2. These transactions are recorded on the L2 blockchain, just like transactions are recorded on Ethereum. But one crucial difference is after a period (say, 10 blocks have passed), there is a process compressing all of these transactions into compressed data, and then this compressed data is recorded on the Ethereum blockchain itself.

That way, if somebody tries to alter the transaction data on the L2 blockchain that has been recorded on Ethereum, you can recompress the data again and compare it with the recorded data on L2.

It might be easy to change the transaction data on the L2 blockchain. An L2 blockchain can only be secured by 10 nodes, but it's very hard to change data on Ethereum, which is secured by 1,000 nodes. This is why people say L2 piggybacks its security to Ethereum.

To put it in a nutshell, users do many transactions on an L2 blockchain. For every period, these transactions are compressed and recorded into Ethereum. So, data on the L2 blockchain is stored on Ethereum, just like transactions on Ethereum are stored in Ethereum. The difference is data from the L2 blockchain is compressed first before being stored in Ethereum.

You can imagine the relationship between Ethereum and L2 as being like what's shown in the following diagram:

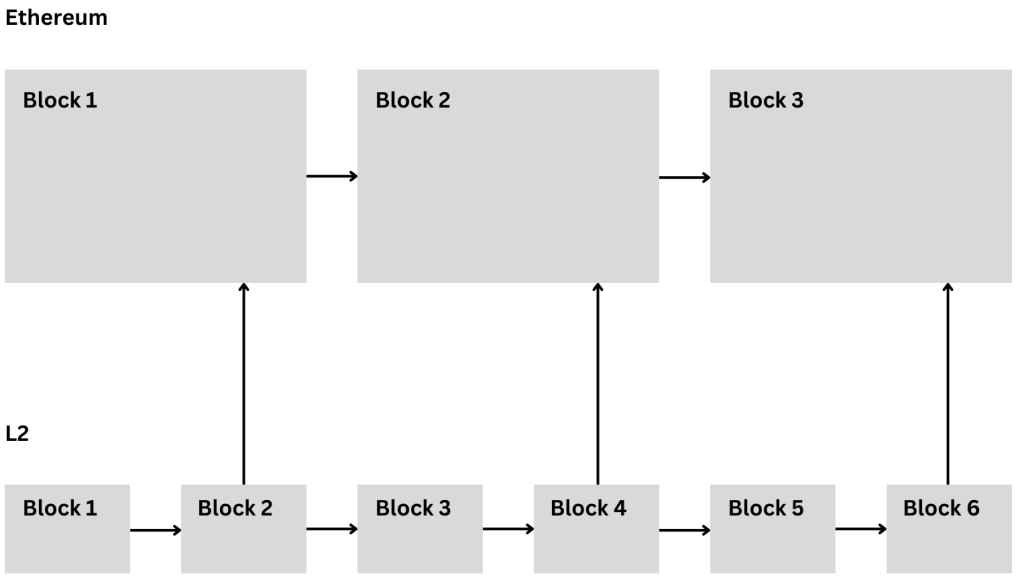


Figure 11.1: Relation between Ethereum and L2 blockchains

How do we store transactions from L2 to Ethereum? There are some smart contracts deployed on Ethereum that have a dedicated purpose to store L2 transaction data into Ethereum. Some L2 nodes have authorization to call methods on these smart contracts.

Transactions happening on Ethereum can be similar to transactions happening on L2 blockchains. You can send ETH to an address on Ethereum. You can also send ETH to an address on an L2 blockchain. You can deploy and interact with a smart contract on Ethereum. You can also do that on an L2 blockchain. You can deploy a token smart contract named USDT on Ethereum. You can also deploy a token smart contract named USDT on an L2 blockchain. An address on Ethereum can have a different balance than an address on an L2 blockchain, although they are exactly the same.

The `0xb7b9a7db7bd0393184cf7aeb080da45a8c0a02b5` Ethereum address and the corresponding address, `0xb7b9a7db7bd0393184cf7aeb080da45a8c0a02b5`, on an L2 blockchain can have distinct states, highlighting the importance of context in understanding the status of an address across different blockchain networks. The address on Ethereum could have 5 ETH in the balance, but the same address on an L2 blockchain could have 10 ETH in the balance.

Both addresses are controlled by the same private key. If you can access `0xb7b9a7db7bd0393184cf7aeb080da45a8c0a02b5` on Ethereum, then you can access `0xb7b9a7db7bd0393184cf7aeb080da45a8c0a02b5` on every L2 blockchain.

As mentioned previously, there can be many L2 blockchains. If there are 100 L2 blockchains on Ethereum and you can access the `0xb7b9a7db7bd0393184cf7aeb080da45a8c0a02b5` address on Ethereum, it means you can access the same address on 100 L2 blockchains. The relation between an address and a private key is the same on Ethereum and L2 blockchains. Because of this relation, Ethereum is referred to as **Layer 1** (**L1**) or the **mainnet** to differentiate between Ethereum and L2.

The term compressing many transactions on an L2 blockchain is known as a **rollup**. The original term "rollup" comes from finance and means buying and combining many small companies into one big company for efficiency. On L2, the cost of storing many transactions (say, 100 transactions) is the same as storing 1 transaction on Ethereum. This fee will be split among users who created those many transactions. Hence, the transaction fee on L2 will be much lower. Of course, there is the additional cost of L2 infrastructure, but it can be kept light.

There can be more than one L2 blockchain on top of Ethereum. You can have as many L2 blockchains as you like. Of course, attracting users to do transactions on the L2 blockchain is another thing.

There are a couple of different types of rollups. But the main ones are **optimistic rollups** and **zero-knowledge (Zk) rollups**.

Optimistic rollups

In this type of rollup, transactions are executed outside the Ethereum mainnet. Then, the transactions are bundled and sent to the Ethereum mainnet. By default, these transactions are assumed to be valid. When nodes are sending batches of transactions to the Ethereum mainnet, we assume all of them are valid. This is why it's called optimistic rollups because we're optimistic about the validity of the transactions. But of course, users can and will cheat. There is a way to make sure cheating doesn't happen. When you are suspicious of certain transactions being invalid, you can challenge it by sending proof of fraud, as long as you do it inside a time window of the challenge period (usually 1 week). If

you are indeed correct, the state of rollups will be recomputed. Invalid transactions will be thrown away. The nodes responsible for including invalid transactions will receive a penalty.

Zk rollups

In this type of rollup, the same as the first type of rollup, transactions are executed outside the Ethereum mainnet. Then, the transactions are bundled and sent to the Ethereum mainnet. But this time, proof of validity is also sent along the way. The smart contract responsible for adding the bundle of transactions from L2 into the Ethereum mainnet will check whether these transactions are valid or not with proof of validity. If not, they will be rejected. So, in a sense, this type of rollup is more paranoid. But on the other side, there is no challenge period. You can always assume every recorded transaction to be valid.

So, with an understanding of L2, let's now take a look at some examples.

Looking at examples of L2

There are many L2 blockchains that are competing to attract users. The most popular ones are Polygon, Arbitrum, Optimism, Manta, Base, Starknet, and zkSync, but there are many more. You can check a list of active L2 blockchains at `https://l2beat.com`. Let's take a look at some examples of L2 blockchains.

Polygon

Although I've explained that L2 is a technology that helps Ethereum scale, there is another technology that has the same goal – this is sidechain. The difference between these two technologies is that sidechain technology doesn't depend on the Ethereum mainnet for its security. Sidechain has its own algorithm to validate transactions. Polygon is a sidechain blockchain. It can interact with the Ethereum mainnet by posting cryptographic messages to a smart contract on the Ethereum mainnet.

So, Polygon shouldn't be on this list because it is a sidechain blockchain. However, its consensus mechanism in validating transactions gets a little help from the Ethereum mainnet. Polygon posts the checkpoint of blocks in the validation layer to the Ethereum mainnet. So, it's also an L2 blockchain. However, it's not an optimistic rollup nor a Zk rollup L2.

Arbitrum

Arbitrum is the most popular L2 on Ethereum at the time of this writing. It's an optimistic rollup L2 blockchain. To deploy smart contracts on the Arbitrum blockchain, the process is similar to deploying smart contracts in the Ethereum mainnet. You can write smart contracts in Solidity and Vyper and deploy them to the Arbitrum blockchain. But Arbitrum gives you the option to write smart contracts in other more familiar languages, such as Rust, Zig, C++, or anything that can be compiled down to WebAssembly. Arbitrum has a WebAssembly **virtual machine** (**VM**) that can execute WebAssembly programs. This technology is called **Stylus**.

Optimism

Optimism is the second most popular L2 on Ethereum at the time of this writing. It's also an optimistic rollup L2 blockchain. It's similar to Arbitrum in the sense that you write smart contracts in Solidity and Vyper and deploy them to the Optimism L2 blockchain. But you cannot write smart contracts in Rust because Optimism doesn't have the Stylus technology. Although Optimism and Arbitrum are optimistic rollup L2 blockchains, they can have some differences in the implementation and innovations on top of the L2 blockchain.

Optimism has released a vision called **Superchain**. It's a network that is composed of many L2 blockchains built using Optimism technology. The Optimism project has released its code, which you can use to build another Optimism-like L2 blockchain for your purpose. This L2 of yours can interact and connect with the Optimism L2 blockchain and another L2 blockchain that's built using Optimism technology. For example, you can send ETH in your L2 to the Optimism blockchain using Superchain.

Base

Base is an L2 blockchain that is built using Optimism technology. It's from Coinbase, the famous crypto exchange in the US. There's not much difference compared to Optimism. It's in the top 10 L2 blockchains based on the market share. Using Superchain, one day, you will be able to interact with the Optimism blockchain from the Base blockchain.

Starknet

Starknet is an L2 blockchain that is built using the ZK-rollup technology. You cannot write smart contracts in Solidity or Vyper anymore. You have to use a different language, which is called **Cairo**. Starknet doesn't use **Ethereum Virtual Machine** (**EVM**). The scaling technology requires a different programming language that maximizes the scalability of Starknet. So, you have to use totally different tools to develop blockchain applications on Starknet.

zkSync

zkSync is another L2 blockchain that is built using the ZK-rollup technology. However, unlike Starknet, you can write smart contracts in Solidity or Vyper because zkSync uses EVM, with some minor differences. So, from the developer's perspective, it's no different than using EVM-compatible L2 blockchains such as Optimism, Arbitrum, or Base. All existing tools can be used.

However, the preceding paragraph covers the newest version of zkSync, called zkSync Era. The first version, which doesn't support EVM compatibility, still runs and is called zkSync Lite.

Now that you've read about examples of L2 blockchains, you can move on to deploying smart contracts to an L2 blockchain.

Deploying smart contracts to L2

For this section, let's deploy a smart contract to the Arbitrum blockchain. But you'll deploy it to an Arbitrum testnet blockchain. At the time of this writing, there are two Arbitrum testnet blockchains. They are the Arbitrum Goerli blockchain and the Arbitrum Sepolia blockchain.

You'll still need to use the same tool that you've used in previous chapters, which is the Ape Framework. Follow these steps:

1. Let's install the framework:

```
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install eth-ape'[recommended-plugins]'==0.7.23
```

2. This time, you add extra plugins because you use Arbitrum and Infura, which is the provider for connecting to the Arbitrum Sepolia testnet blockchain:

```
(.venv) $ pip install ape-infura ape-arbitrum
```

3. Then, you initialize the project directory:

```
(.venv) $ mkdir arbitrum
(.venv) $ cd arbitrum
(.venv) $ ape init
```

You can fill in `ArbitrumContract` as the name of the project when prompted.

Then, you create a new file under the `contracts` folder named `SimpleStorage.vy` and add the following code to it:

```
# @version ^0.3.0

num: uint256

@external
def store(num: uint256):
    self.num = num

@external
def retrieve() -> uint256:
    return self.num
```

Then, you compile it by using the following command:

```
(.venv) $ ape compile
```

Then, you create a deployment script in the `scripts/deploy.py` file by putting the following code into the file:

```
from ape import accounts, project
import os


def main():
    password = os.environ["MY_PASSWORD"]
    dev = accounts.load("dev")
    dev.set_autosign(True, passphrase=password)
    contract = project.SimpleStorage.deploy(sender=dev)
    num_value = contract.retrieve.call()
    print(f"The num value is {num_value}")
```

Make sure you have an Ape account named `dev` before you want to use the deployment script. Also, you need to set the password for the `dev` account to the `MY_PASSWORD` environment variable.

If you haven't created an Ape account, you can do so this way:

```
(.venv) $ ape accounts generate dev
```

You can export the password to the environment variable this way:

```
(.venv) $ export MY_PASSWORD=yourpassword
```

Before you deploy the smart contract to the Arbitrum Sepolia testnet, you need a way to connect to the Arbitrum Sepolia testnet blockchain or network. You can run a full Arbitrum node so that you can connect to this node when deploying smart contracts. The downside of running a full node is the high requirement. You need 8-16 GB RAM, and large storage (half a terabyte on SSD storage). This is a high barrier to entry. On top of that, you also need access to an Ethereum node, which requires more memory and storage than an Arbitrum full node.

The good news is you don't have to do that. You can use a service from one of the node providers. You've used the service from Alchemy in previous chapters. This time, you'll use another node provider, which is Infura.

You need to register an account in Infura at `https://infura.io`. Then, go to the dashboard page and you'll be presented with this page:

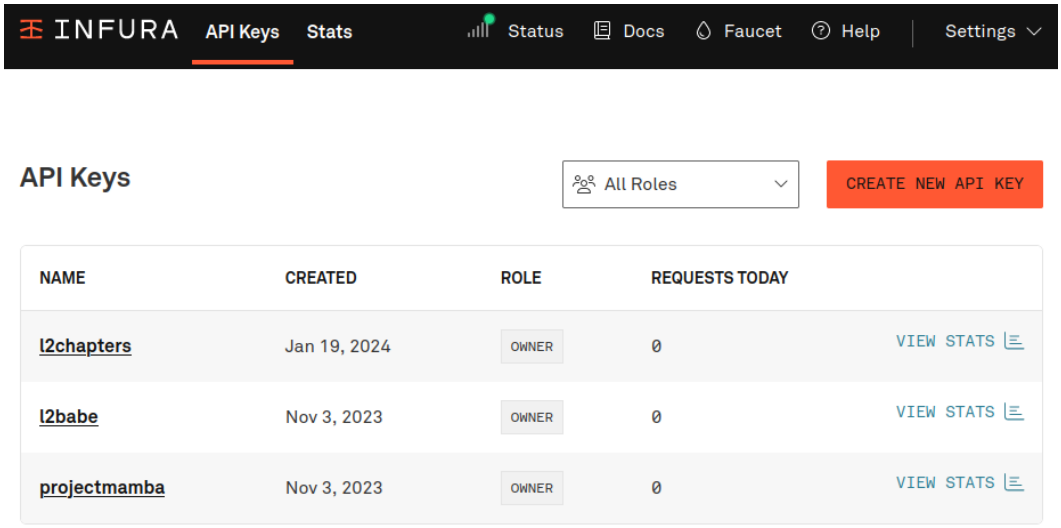


Figure 11.2: Infura dashboard

Click the **CREATE NEW API KEY** button, and you'll be presented with a pop-up window:

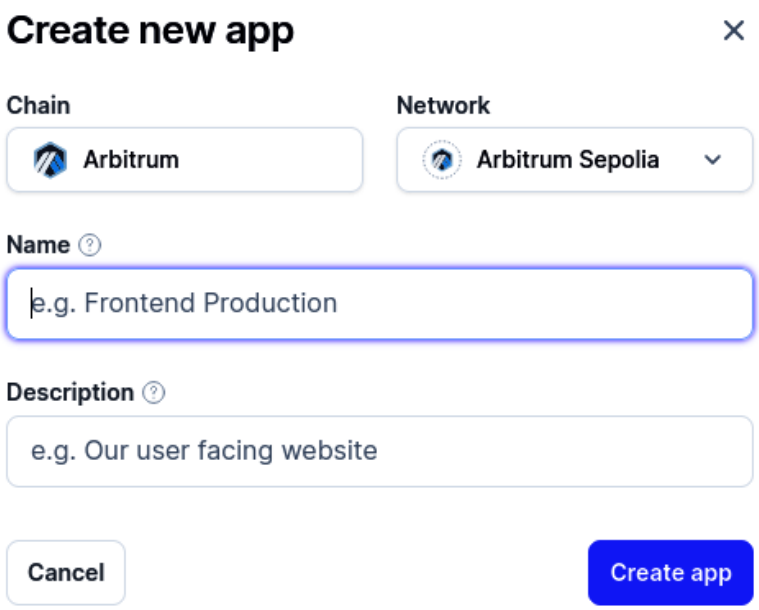


Figure 11.3: The Create new app window

Choose **Arbitrum** in the **Chain** field and choose **Arbitrum Sepolia** in the **Network** field. Fill in whatever name you like for this application. Finally, click the **Create app** button. You'll get the new API key. You can click the name of the new API key on the dashboard page, then you'll be presented with the API key that you can use, as presented in the next figure:

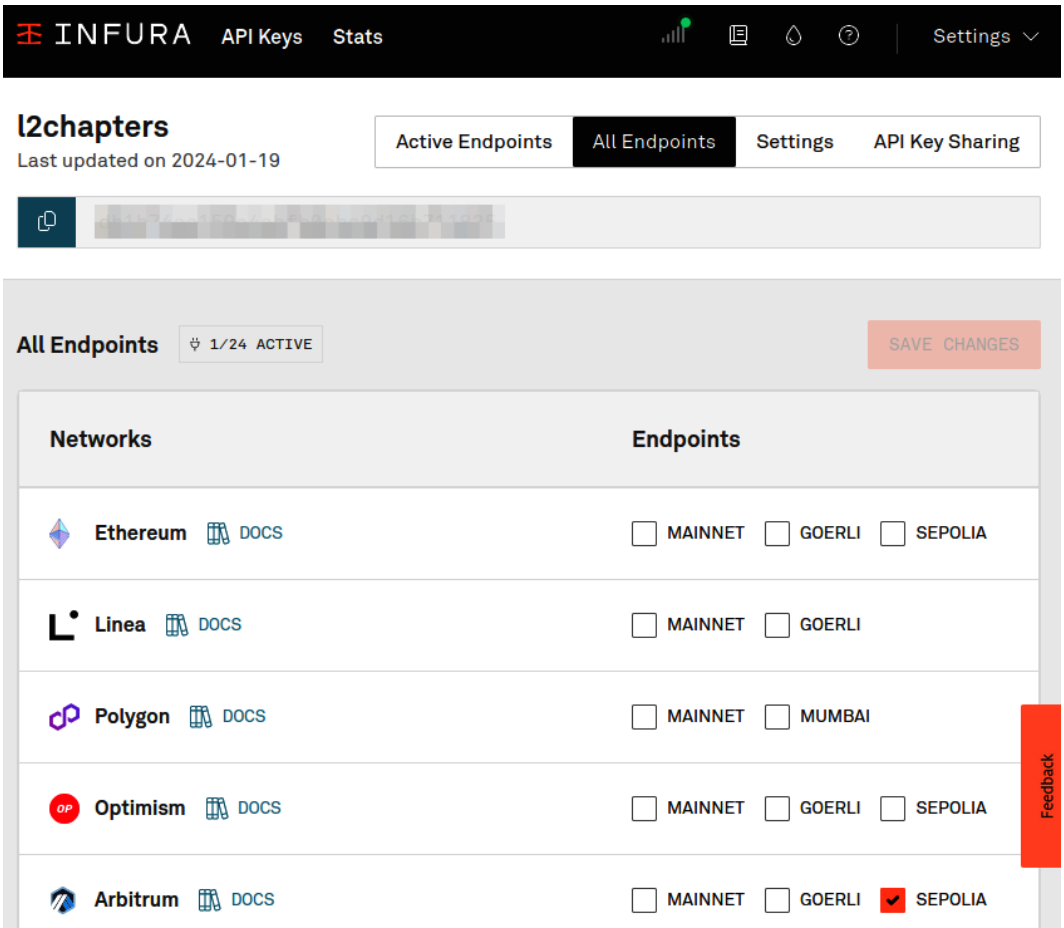


Figure 11.4: The API key page

On the API key page, you can also set whether this API key can be used for certain chains. For your purpose, you want to make sure **Arbitrum Sepolia** is checked.

Copy the API key and put it in the `WEB3_INFURA_API_KEY` environment variable:

```
(.venv) $ export WEB3_INFURA_API_KEY=theAPIkeyfromInfura
```

To deploy a smart contract in Arbitrum Sepolia, you need some ETH. You can find them on the Faucet website at `https://arbitrum-faucet.com/`. To get some free ETH, you must log in to this website using an Alchemy account. You can register an account on the Alchemy website at `https://alchemy.com`. Put your `dev` Ape account address in the field, then click the **Send Me ETH** button:

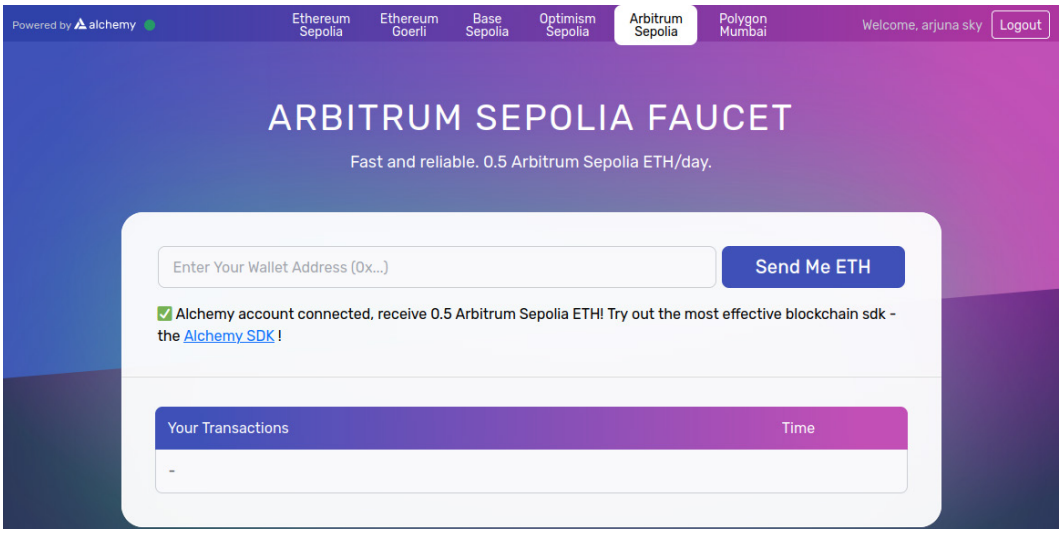


Figure 11.5: The Arbitrum Sepolia Faucet website

There are other faucet websites, such as `https://faucet.triangleplatform.com/arbitrum/sepolia`, `https://faucet.quicknode.com/arbitrum`. Once your account receives some ETH from the Faucet website, you can launch your smart contract to the Arbitrum Sepolia website:

```
(.venv) $ ape run deploy --network arbitrum:sepolia:infura
WARNING: Danger! This account will now sign any transaction it's
given.
WARNING: Using cached key for dev
INFO: Submitted
0xb0803dbe51d81474a7f8102d007c40c84f4f8003b5e61ec65577e3826b2e7b3f
Confirmations (1/1): 100%|████████████████████████| 1/1
[00:00<00:00,  3.31it/s]
INFO: Confirmed
0xb0803dbe51d81474a7f8102d007c40c84f4f8003b5e61ec65577e3826b2e7b3f
(total fees paid = 8402700000000)
SUCCESS: Contract 'SimpleStorage' deployed to:
0xe10D464ff17681Bcad550716671797eC3cd79b07
The num value is 0


https://sepolia.arbiscan.io/
address/0xe10d464ff17681bcad550716671797ec3cd79b07
```

The last line is the URL where you can check the transaction hash of your smart contract deployment, as shown in the following figure:

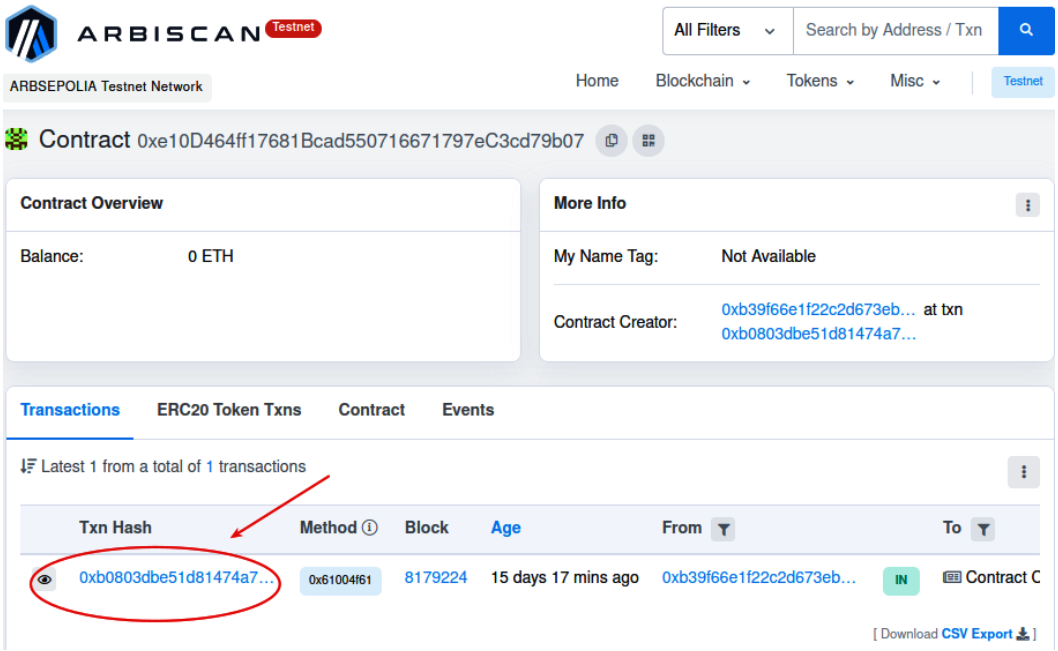


Figure 11.6: The Arbiscan website

If you click the transaction hash, you'll get the status of your smart contract deployment, as shown in the following figure:

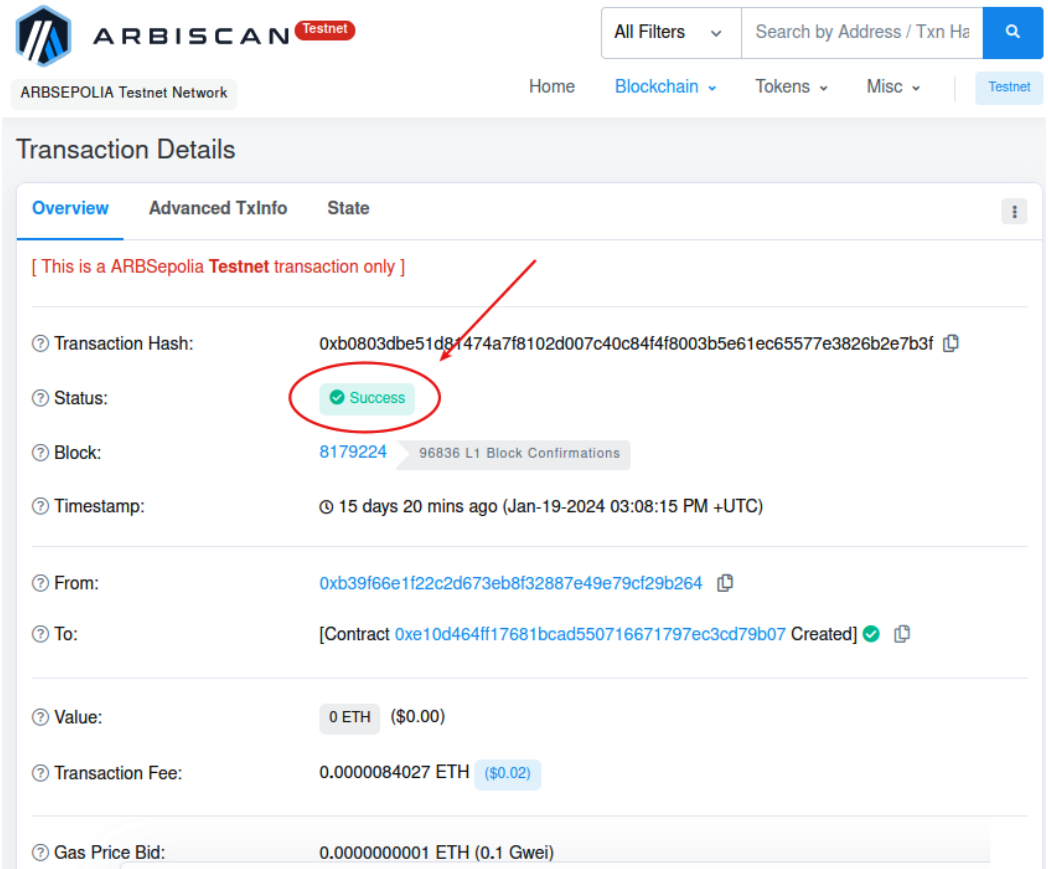


Figure 11.7: The status of the transaction

Deploying smart contracts to other L2s is similar to this way. Most of the time, you wouldn't set up L2 nodes directly. You would use infrastructure providers such as Infura or Alchemy, then you can use their APIs to connect to L2 blockchains. These providers provide many blockchains that you can use directly.

Now that you've deployed a smart contract to L2, it's time to learn how to interact with the Ethereum mainnet from L2.

Introducing the bridge technology

Validators earn the ETH currency as a reward when validating transactions on the Ethereum mainnet. So, does that mean ETH does not exist on L2s such as Arbitrum, Optimism, Starknet, and so on? ETH does exist on L2s. ETH are not created from validators of transactions on L2s. ETH on L2 comes from ETH on the Ethereum mainnet. So, how do you send ETH from the Ethereum mainnet to an L2?

Bridging ETH

Basically, an L2 blockchain has smart contracts that live on the Ethereum mainnet that act as a gateway or connector between the Ethereum mainnet and L2. The way it works is simple. On the Ethereum mainnet, you send X amount of ETH to a smart contract called Y. Then, X amount of ETH will be born on the L2. The amount of ETH on the L2 blockchain tallies with the amount of ETH that the Y smart contract on the Ethereum mainnet holds.

But to whom does this ETH belong? It's up to the rules you define in the smart contract. You can create a `map` variable in the smart contract that maps an account to its balance of ETH on L2. Then the method of the smart contract that you send ETH to can query the address sender and set the balance of ETH in the `map` variable. You can imagine how it works in the following figure:

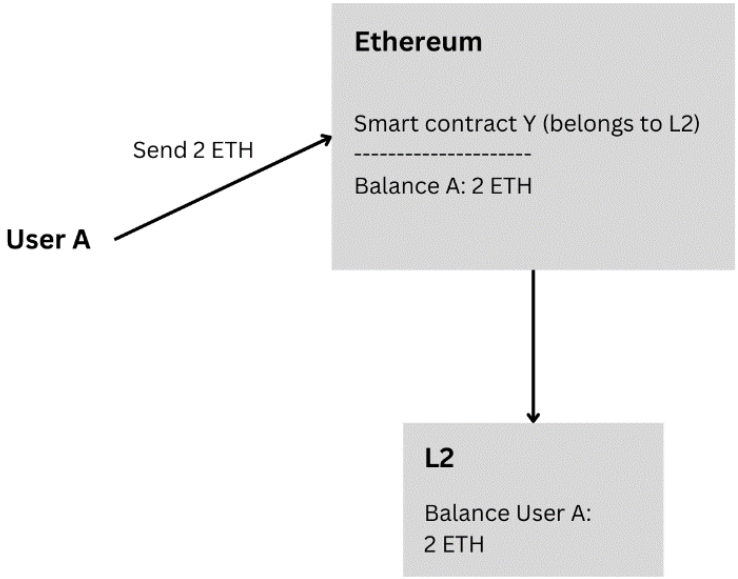


Figure 11.8: Bridging ETH to L2

You can do whatever you want with your ETH on L2. If you withdraw ETH from L2, it means you withdraw ETH from the smart contract in the Ethereum mainnet to your wallet or another address. The amount of ETH on the smart contract decreases, and the amount of ETH on the L2 decreases. The amount of ETH that decreases should be the same between the two.

To put it in a nutshell, the amount locked in the smart contract is the same as the amount of ETH in the L2 blockchain. Each L2 blockchain has its own smart contract that holds ETH.

Bridging ERC-20 tokens

It's not just ETH that you can bridge to L2 from the Ethereum mainnet. You can also bridge ERC-20 tokens to L2. The way you do it is similar to bridging ETH. You send ERC-20 tokens (for example, USDT) to a smart contract that acts as a gateway on the Ethereum mainnet. You lock USDT into the smart contract. Then, the same amount of USDT will exist on the L2 blockchain.

To whom does this USDT belong? It's up to the rules you define in the smart contract. Just like in the bridging ETH to L2 case, you can define a `map` variable that defines an account to its balance of USDT or other ERC-20 tokens.

However, the amount of USDT in the smart contract must tally with the amount of USDT in the L2 blockchain.

Native ERC-20 tokens

However, for ERC-20 tokens, you don't have to follow the bridging way. You can create ERC-20 tokens natively on L2. Remember – you can do anything on L2 just as you do things on Ethereum. You can deploy a smart contract and execute the method of a smart contract. You can send ETH to an address. It means you can also deploy an ERC-20 token smart contract on L2.

There is the USDC smart contract on Ethereum deployed by the Circle company. 1 USDC on the Ethereum mainnet can be redeemed with 1 USD. So, when the Arbitrum blockchain was released, users wanted to use USDC, so they used bridging. With this, a user sends 100 USDC on the Ethereum mainnet to a bridge smart contract that will lock this 100 USDC. The user then can use 100 USDC on Arbitrum.

But Circle later deployed the USDC smart contract on Arbitrum, and the company mints USDC natively. So, you can get USDC from their smart contract and don't have to use bridging anymore. This is in point of making Arbitrum a totally complete blockchain while still being tied to the Ethereum mainnet.

Messages

Using the smart contract is also the way the L2 blockchain sends data on the state of transactions happening in L2. Usually, not everyone can call methods to update the state of L2 transactions because the state in L2 is paramount to the integrity of L2. For example, there are a couple of nodes that can update the state of an L2 blockchain on the Ethereum mainnet. When you create a transaction on L2, these nodes are responsible for validating your transactions and later sending them along with other transactions to the smart contract on the Ethereum mainnet. The smart contract will save data into the Ethereum mainnet. The saving action is triggered by nodes calling certain methods of the smart contract. Other people cannot call these methods.

In a way, this protects random people from ruining the integrity of L2, but the downside is there is a centralization threat. Suppose there are nine nodes responsible for recording L2 transactions on the Ethereum mainnet, and those nodes are owned by one person. This one person can stop L2 transactions from being recorded on the Ethereum mainnet. You can create transactions on L2 but they will not be validated nor recorded on the Ethereum mainnet.

So, L2 cannot stand alone without the Ethereum mainnet, and their connection is smart contracts on the Ethereum mainnet.

This is just a general view of how nodes send and receive messages from and to L2. The details of how nodes send and receive messages differ, of course.

To look at one of the L2 blockchains, in Arbitrum, at the time of this writing, two kinds of nodes are responsible for submitting transactions and publishing them to the Ethereum mainnet:

- The first one is Sequencer, which will receive L2 transactions, reorder them, execute them in the Arbitrum VM, and then give receipts to users who created and submitted the transactions. These Sequencer nodes are centralized.
- Then, the second kind of node in Arbitrum is Validator. They are there to validate the state of transactions and make sure no one is cheating. Right now, to become a validator in Arbitrum, you must get inside the allow list.

Sending messages from L1 to L2

An L2 blockchain provides ways for you to send messages from L1 to L2 (for example, depositing ETH to L2 from the Ethereum mainnet) and send messages from L2 to L1 (for example, withdrawing ETH to the Ethereum mainnet from L2). A message doesn't have to be withdrawing ETH or depositing ETH. It can be a static string that's being recorded on the blockchain. It can also be an attempt to execute a method on the smart contract.

In Arbitrum, to send a message from the Ethereum mainnet or L1 to L2, you must create a **retryable** ticket. The word "retryable" implies the request in a message will be tried if something happens. Let's explore the details of the retryable ticket.

The Arbitrum team has deployed a contract named `Inbox` on the Ethereum mainnet. To send a message to L2, you can execute the `createRetryableTicket` method of the `Inbox` smart contract. This method accepts a couple of arguments. For example, it accepts the destination address on L2 (where you want to send the message from L1) and the data (the message). These arguments make sense. That's the point of sending messages.

There is another argument, such as the maximum ETH that you want to use when creating a retryable ticket on L1. This also makes sense because any transaction, such as executing a method of a smart contract on the Ethereum mainnet, requires a fee. It's not free.

There are also other arguments that are related to L2, such as `gasLimit` and `maxFeePerGas`. This is similar to the gas limit and fee when you executed the method of a smart contract in previous chapters. Creating transactions on Arbitrum is not free either (but it's much cheaper compared to L1).

But there is an argument that gives you an option to set the address to receive a refund of the remaining ETH you get when creating a retryable ticket. There is also an argument that you can use to put an address to receive a refund of ETH you get when the retryable ticket is rejected.

To make this clear, let's create an example. When creating a retryable ticket, you also send 0.1 ETH to the method. To create the transaction on L1, you pay 0.06 ETH. Then on L2, a transaction of sending the message will be created. It's not free, but you still have 0.04 ETH to pay a fee for creating the transaction on L2. Suppose the fee of creating the transaction on L2 is 0.01 ETH; then, it means you have 0.03 ETH remaining. Arbitrum will refund 0.03 ETH to the address you defined to receive the refund of the remaining ETH.

To make things interesting, it's not 100% sure that the transaction on L2 will be created. Suppose you create a retryable ticket on L1 that is going to say `"Good Morning"` to an address on L2. The retryable ticket has been created. But that doesn't mean the `"Good Morning"` message will be sent on L2. It could fail for whatever reason. When it fails, Arbitrum will send back ETH that you provide to an address you set. The ticket lives for a period of time (such as 1 week). In this period, you have an option to do manual redemption, meaning to try to create a transaction on L2 again, which sends the `"Good Morning"` message. To redeem, you can call the `redeem` method of `ArbRetryableTx`.

Sending messages from L2 to L1

Sending a message from L2 to L1 requires a similar procedure to sending a message from L1 to L2. But there are some fundamental differences.

Remember – for every period of time, Arbitrum (or L2 blockchains in general) will put the state of the transactions of L2 in L1. The smart contract on L1 that receives this data is the `Outbox` contract. The name indicates the flow of the message from L2 (inner) to L1 (outer). Once Arbitrum sends data or messages from L2 to the `Outbox` smart contract, it's not done yet. You need someone to execute the messages on L1. In this context, executing messages means validating transactions on L2.

From the user's point of view on L2, to send a message to L1, you need to execute the `sendTxToL1` method of the `ArbSys` smart contract. The message will be confirmed in L2 in a time period of 1 week. Then, you can execute the message on L1 so that it is actually sending messages from L2 to L1.

So, there are two steps: executing the method of the smart contract on L2, waiting for 1 week, and then executing the method of the smart contract on L1. Then, the message from L2 to L1 is actually sent.

However, you must remember that these steps only happen on Arbitrum. The other L2 blockchains may have different procedures for sending messages from L2 to L1. In ZK-rollup L2 blockchains such as Starknet, you don't need to wait for 1 week to send a message. Sending a message from L2 to L1 can be confirmed in a short time.

The other difference between sending messages from L2 to L1 and from L1 to L2 is that there is no expiry date for a message to be executed on L1. There is no scheduled execution of messages on L1 like the one in L2. This is why sending messages from L1 to L2 is called *creating a retryable ticket*, but not in the other case.

Sending messages across L2s

What if you want to send a message from Arbitrum to a different L2 blockchain, such as Optimism or Starknet? As a reminder, sending a message can mean sending ETH, transferring tokens, or voting in a smart contract.

Let's take an example of sending ETH from Arbitrum to Starknet. The way to do it is you withdraw ETH from Arbitrum to the Ethereum mainnet and then wait 1 week. Then, you deposit ETH from the Ethereum mainnet to Starknet.

The Ethereum mainnet becomes the hub between L2s. Every message you want to send from an L2 blockchain to another L2 blockchain needs to pass the Ethereum mainnet first. This makes sense because L2 posts data regularly to the Ethereum mainnet for transaction integrity.

The problem, of course, is it's not practical and takes too long. You want to send messages or send ETH in a short time from one L2 to another L2. But there is a way to do that. Imagine you have 10 ETH on Arbitrum and 10 ETH on Optimism and I want to send 2 ETH from Arbitrum to Optimism. The way to do it is I send 2 ETH to you in Arbitrum so that your end balance in Arbitrum is 12 ETH, then you can send 2 ETH to my address on Optimism, so your end balance in Optimism is 8 ETH. You can imagine the process as shown in the following figure:

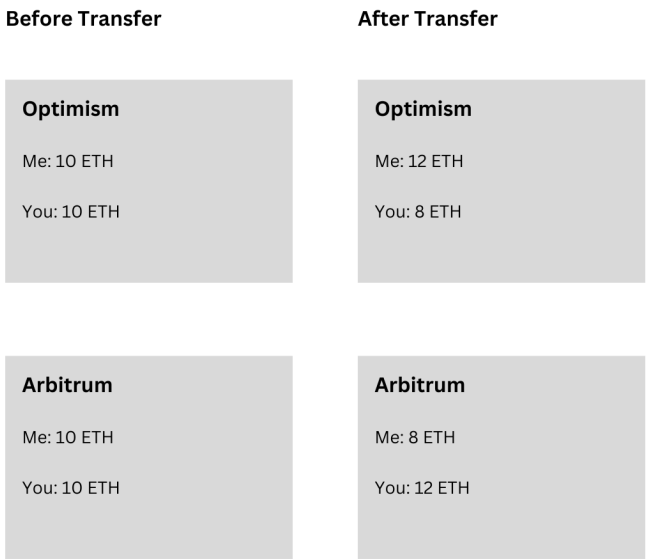


Figure 11.9: The concept of multi-chain transfers

In using this approach, some people need to provide money (ETH or ERC-20 tokens) first in the destination blockchains. The money that is being provided is called liquidity, and people who provide this money are called market makers.

To make this approach work, you need a lot of liquidity. To attract market makers, you need to give an incentive. The most attractive incentive is profit. You can get profit from the fee that you charge to users who want to use this bridge.

To make sure each L2 destination has enough liquidity, you can create an incentive such that the profit for market makers for L2 destinations that lack liquidity is much higher.

Sending messages across blockchains

You've seen how you can create a way to send messages (sending tokens or ETH) between L2 blockchains on Ethereum. But who says you cannot implement a similar way to send messages across totally different blockchains, such as sending messages from Ethereum to Solana?

A similar procedure could apply. You set up liquidity on the Ethereum mainnet and Solana. You have 10 ETH on the Ethereum mainnet and 10 ETH on Solana. I want to send 2 ETH from the Ethereum mainnet to Solana. So, I send 2 ETH to you on the Ethereum mainnet so that your end balance on the Ethereum mainnet is 12 ETH. Then, you can send 2 ETH to me on Solana so that your end balance on Solana is 8 ETH.

Such cross-chain applications are many. Some examples are Wormhole, (`https://wormhole.com`) and LayerZero (`https://layerzero.network`). LayerZero supports more than 50 blockchains at the time of this writing. Wormhole supports more than 20 blockchains at the time of this writing.

The nice thing about using cross-chain applications such as LayerZero or Wormhole is you get unified functions to interact with many blockchains. To send a message with LayerZero from a blockchain to another blockchain, you can call the `_lzSend` method. To receive a message with LayerZero in a blockchain from another blockchain, you can call the `_lzReceive` method. It doesn't matter from which blockchain and to which blockchain you want to send a message. You use the same function. If you want to send a message from Solana to BNC, you use `_lzSend` on Solana and `_lzReceive` on BNC. But if you want to send a message from the Ethereum mainnet to Optimism, you use `_lzSend` on the Ethereum mainnet and `_lzReceive` on Optimism. Cross-chain applications such as LayerZero have abstracted the functions you need to use. You don't need to know the details.

If you are an astute reader, you'll notice that there is a security issue with cross-chain applications such as LayerZero. Say I want to send 2 ETH from Arbitrum to Optimism; you act as the cross-chain application, so you provide liquidity on Arbitrum and Optimism. In the first step, I send 2 ETH to you on Arbitrum. I expect you to send 2 ETH (minus the fee for your profit) to me on Optimism. Everyone will be happy in the end. I can transfer ETH from Arbitrum to Optimism. You get some profit as you charge a fee to me.

But what if you don't honor the agreement? You receive 2 ETH on Arbitrum, but you don't send 2 ETH on Optimism to me.

This risk is the same as the risk when you use a centralized exchange such as Coinbase or Binance. What makes sure that they don't steal your funds? Only the law and the goodwill of the administrators of centralized exchanges.

However, in blockchain, people want to avoid trusting centralized actors as much as possible. Ideally, blockchain applications are decentralized and secure. But how can you make cross-chain applications secure?

You can decentralize the actors that act as a bridge. Suppose you create a box that will receive ETH from a source address on a source blockchain and then send ETH to the destination address on a destination blockchain. If the box is only operated by one person, yes – it's centralized and has a high-security risk. But what if there are many people who operate that box? And what if these people are not in the same entity or – put another way, it's decentralized? You can imagine "these people" as another blockchain that has sufficient decentralization. That way, you can reduce the risk in a bridge or cross-chain application.

In Hop Exchange (`https://hop.exchange`), a bridge application where you can send tokens across L2s, the way they do it is to create some kind of roll-up application. Say you want to send 2 ETH from Arbitrum to Optimism. You change 2 ETH with 2 Hop ETH on Arbitrum. You use a Hop bridge to send 2 Hop ETH from Arbitrum to Optimism. You receive 2 Hop ETH on Optimism. Then, you swap 2 Hop ETH with 2 ETH on Optimism.

This transfer transaction will be recorded along with other transfer transactions to the Ethereum mainnet through an L1 Hop bridge. The L1 Hop bridge will collect batches of transfer transactions and record them to the Ethereum mainnet. This way, the integrity of transactions of the Hop bridge can be secured on the Ethereum mainnet.

The knowledge of bridge applications enables you to move values in and out of many blockchains. It is unlikely that only blockchain will be used by all people. We live in a multichain era, and it's useful to know how to connect between blockchains.

Summary

In this chapter, we learned the limitations of Ethereum, which are scalability and high fees. Then, we saw that L2 was built to mitigate these issues. We also learned how L2 works and its relation to Ethereum. After that, we looked at examples of L2s, such as Optimism, Arbitrum, Base, Starknet, and so on. Then, we deployed a smart contract to the Arbitrum testnet blockchain. In the end, we learned about bridge applications and how we move data between Ethereum and L2. The lessons you learned in this chapter enable you to build decentralized applications with lower fees and more scalability while still depending on Ethereum security. In the next chapter, we will learn about the ERC-20 token standard and how to create tokens.

Part 5: Cryptocurrency and NFT

This section immerses you in the dynamic intersection of cryptocurrencies and non-fungible tokens (NFTs). You will master the art of tokenization, unlocking the power to represent various assets on the Ethereum blockchain. Further, you will journey into the realm of digital ownership by learning the intricacies of minting and trading non-fungible tokens and, in the process, shaping the future of digital collectibles and beyond.

This section has the following chapters:

- *Chapter 12*, *Creating Tokens on Ethereum*
- *Chapter 13*, *How to Create an NFT*

12
Creating Tokens on Ethereum

In this chapter, you are going to learn how to create tokens on top of Ethereum. Tokens have a variety of uses; they may represent a local currency in a local community, a physical good, virtual money in a game, or loyalty points. You can build new cryptocurrencies with this token; Ethereum is a cryptocurrency itself, but you can build new cryptocurrencies on top of it, making it far easier to create a new token.

This chapter will cover the following topics:

- How to create a simple token smart contract
- The ERC-20 token standard
- Selling tokens
- Security aspects

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_12`.

How to create a simple token smart contract

Tokens are the real killer applications on Ethereum. Tokens can represent many things, from virtual currency, loyalty points, shares in a company, or a venture into a representative of real-world assets. The most famous application of tokens is stablecoins, which represent fiat currency, usually US Dollars. Examples are USDC and USDT, where 1 USDC represents 1 US Dollar fiat. You can redeem 1 USDC with 1 US Dollar.

Without further ado and to help develop our notion of what a token is and how it works, let's create a really simple token smart contract.

Let's install Ape Framework:

```
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install eth-ape'[recommended-plugins]'==0.7.23
```

Then, create the project directory:

```
(.venv) $ mkdir token_project
(.venv) $ cd token_project
```

After creating it, initialize the project:

```
(.venv) $ ape init
Please enter project name: HelloToken
SUCCESS: HelloToken is written in ape-config.yaml
```

Create a new file inside the `contracts` folder and name it `VerySimpleToken.vy`. Then, add the following code to it:

```
#pragma version ^0.3.0

balances: public(HashMap[address, uint256])

@external
def __init__():
    self.balances[msg.sender] = 10000

@external
def transfer(_to: address, _amount: uint256) -> bool:
    assert self.balances[msg.sender] >= _amount
    self.balances[msg.sender] -= _amount
    self.balances[_to] += _amount
    return True
```

The code is simple. We have the `balances` state variable that maps an address to an integer. In the `__init__` initialization function, the smart contract gives the key deployer a 10,000 integer value in the `balances` state variable. What does this mean? It means that the deployer, in the beginning, has 10,000 tokens. Other addresses have zero tokens.

Then, by using the `transfer` method, a user can send tokens to another user. The method has the `_to` argument, which represents the destination address, and the `_amount` argument, which represents the number of tokens to be transferred.

The transfer action constitutes decreasing the balance of the caller of the method (represented by `msg.sender`) and increasing the balance of the `_to` destination address. The action is represented by these two lines of code:

```
    self.balances[msg.sender] -= _amount
    self.balances[_to] += _amount
```

The amount to be increased and decreased must be the same because the total number of tokens is 10,000, as stated in the initialization method.

Don't forget to compile the smart contract:

```
(.venv) $ ape compile
```

Let's create tests for this smart contract. Create a new file named `conftest.py` inside the `tests` folder and then add the following code to it:

```
import pytest


@pytest.fixture
def deployer(accounts):
    return accounts[0]

@pytest.fixture
def contract(deployer, project):
    return deployer.deploy(project.VerySimpleToken)
```

This is the helper file for the unit test.

Then, create a unit test file named `test_VerySimpleToken.py` inside the `tests` folder. Then, add the following code to the test file:

```
def test_balances(contract, deployer):
    balance = contract.balances(deployer)
    assert balance == 10000

def test_transfer(contract, deployer, accounts):
    destination_account = accounts[1]
    balance = contract.balances(destination_account)
    assert balance == 0
    balance = contract.balances(deployer)
    assert balance == 10000
    contract.transfer(destination_account, 200, sender=deployer)
    balance = contract.balances(destination_account)
```

```
    assert balance == 200
    balance = contract.balances(deployer)
    assert balance == 9800
```

Using the `test_balances` method tests the initial state of the smart contract, which is the balance of the smart contract's deployer, which is 10,000 tokens.

Using the `test_transfer` method tests the transferring action. Before the transfer takes place, you need to check the balance of the source address and destination address. After the transfer takes place, you need to check them again. For example, the source address balance might have decreased by 200, from 10,000 to 9,800. The destination address balance will have increased by 200, from 0 to 200.

To execute the test file, run the following command:

```
(.venv) $ py.test tests/test_VerySimpleToken.py
============================================== test session starts
==============================================
platform darwin -- Python 3.11.6, pytest-7.4.4, pluggy-1.4.0
rootdir: /Users/arjuna/Code/book/Hands-On-Blockchain-for-Python-
Developers--2nd-Edition/chapter_12/erc20
plugins: web3-6.15.1, eth-ape-0.7.7
collected 2 items

tests/test_VerySimpleToken.py ..
                                                    [100%]

=============================================== 2 passed in 0.54s
==============================================
```

So, what's the big deal about this simple smart contract?

This simple smart contract digitally creates 10,000 token coins and gives all of them to the owner of the smart contract. Then, the owner can forward the coins to other accounts using the `transfer` method.

This simple smart contract is special in comparison to creating a simple traditional web application token because once this smart contract is deployed, the owner cannot change the number of tokens, no matter how desperate they are. If the owner has just used the `transfer` method to transfer some coins to another account's address, they cannot take them back again. Other people can verify the rules of the play in the smart contract before interacting with it.

Compare this with the simple token we created in the traditional web application. Once you create 10,000 coins here, you can still change the number of tokens by updating the number of coins in the database. Additionally, you can change the rule as you like, which places other people who want to interact with this application at a disadvantage.

This token application is the most used type of application in blockchain technology. Blockchain has transparency and censorship resistance properties, which are useful for token applications that deal with values. What I mean by "values" is things such as money. Examples of applications that deal with values are banking applications that need to track the balances of users.

It's very important that the values are not corruptible. For example, it is understandable if only the video aspect of a video game is corrupted, in which case, there is not much of an issue. However, when talking about values in terms of money, people don't take it lightly when such values are stolen or corrupted.

The thing about smart contracts is that they're very flexible. So, you can create any rules that you like. In the previous example of a smart contract, the number of tokens was fixed by a hardcoded number in the initialization method. However, nothing can stop you from adding any rules, such as the ability to add new tokens to existing tokens. Perhaps it will be useful to simulate inflation.

Create a new file named `TokenWithInflation.vy` inside the `contracts` folder. Then, add the following code to it:

```
#pragma version ^0.3.0

balances: public(HashMap[address, uint256])
owner: public(address)

@external
def __init__():
    self.balances[msg.sender] = 10000
    self.owner = msg.sender

@external
def transfer(_to: address, _amount: uint256) -> bool:
    assert self.balances[msg.sender] >= _amount
    self.balances[msg.sender] -= _amount
    self.balances[_to] += _amount
    return True

@external
def mint(_new_supply: uint256):
    assert msg.sender == self.owner
    self.balances[msg.sender] = _new_supply
```

Notice that there is a new state variable named `owner` that has the address data type. In the `initialization` method, this variable is set to the smart contract deployer's address. This refers to the owner of the smart contract.

Then, by using the `mint` method, you can make sure that only the owner of the smart contract can call this method. The method accepts a new supply argument. The balance of the owner is set to this new supply. In other words, you can change the balance of the owner of the smart contract on your whim.

However, does this defeat the purpose of the fairness of blockchain? Again, it's up to people to create rules for smart contracts. You can create an unfair smart contract by adding this `mint` method, whereby you can add any number of tokens to your balance. It's up to you. Smart contracts are very flexible because you can code any rules you want.

However, blockchain gives people transparency. People can see that the `mint` method exists and that it can be used to increase the balance of the owner of the smart contract. It's up to people whether they want to participate in this smart contract or not. The reason why blockchain technology is valuable is because of transparency. You can let people see the rules of the smart contract.

Also, the ability to increase tokens can be viewed positively. This is called inflation. Many countries can print money. If this action is not done excessively, people can still accept it. People can participate in this smart contract and check whether you mint tokens excessively. If people are not comfortable with your actions, they can always choose not to participate in your smart contract.

Although you can write smart contracts freely, there is a standard that you can use when writing token smart contracts.

The ERC-20 token standard

It is more than likely that you will have heard of ERC-20. When a new cryptocurrency is released, the first question that usually arises is, Is it an ERC-20 token? People incorrectly assume that an ERC-20 token is a cryptocurrency based on Ethereum. Well, technically speaking, this is true, but it does not tell the whole story. ERC-20 is a standard for creating a token on Ethereum. The simple token that we have just created does not fulfill the ERC-20 standard. Yes, it is a digital token smart contract, but it is not an ERC-20 token. The ERC-20 standard is one of many reasons why we have seen an increase in the number of new cryptocurrencies. However, ERC-20 is not a requirement for creating a token on top of Ethereum.

To create an ERC-20 token smart contract, you must implement the following methods in your smart contract:

```
function totalSupply() public view returns (uint256)
function balanceOf(address _owner) public view returns (uint256
balance)
function transfer(address _to, uint256 _value) public returns (bool
success)
function transferFrom(address _from, address _to, uint256 _value)
public returns (bool success)
function approve(address _spender, uint256 _value) public returns
(bool success)
function allowance(address _owner, address _spender) public view
```

```
returns (uint256 remaining)
event Transfer(address indexed _from, address indexed _to, uint256
_value)
event Approval(address indexed _owner, address indexed _spender,
uint256 _value)
```

The preceding methods use the Solidity syntax. So, you should convert them to the proper Vyper syntax. If you implement those methods in a smart contract, your smart contract follows the ERC-20 standard.

However, there are optional methods that you can implement, too, which are given in the following code block:

```
function name() public view returns (string)
function symbol() public view returns (string)
function decimals() public view returns (uint8)
```

So, what is so special about this ERC-20 standard? Is this an obligation when creating a token smart contract? Why can't we create a digital token without fulfilling the ERC-20 standard?

Actually, you don't have to follow this standard; there is no law that forces you to create an ERC-20 token. For example, the ERC-20 standard expects you to tell users the overall amount of tokens created using the `totalSupply` method. However, you could create a method named `taylorSwiftIsTheBest` to return the total supply of the tokens, and then you could create a document to explain this method.

However, there are some advantages if you follow the ERC-20 token standard:

- First, it makes it easier for users to audit your smart contract.
- Second, your ERC-20 token will be recognized automatically by an Ethereum wallet, such as MetaMask.
- Third, it is easier for a cryptocurrency exchange to list your token. Basically, it makes everyone's lives easier.
- Fourth, the standard makes it possible for decentralized exchange smart contracts to let ERC-20 tokens be sold and bought.

However, you should treat the ERC-20 standard as guidance. You don't have to follow the ERC-20 standard 100%, but it's a good idea to follow it. Not all popular tokens built on top of Ethereum are 100% ERC-20-compliant. One such example is the Golem token smart contract. The smart contract does not implement the `approve` method, among other things. You can read the source code of the Golem token smart contract at the following link: `https://etherscan.io/token/0xa74476443119A942dE498590Fe1f2454d7D4aC0d#readContract`.

- Having said that, let's create an ERC-20 token smart contract. Create a new file named `HelloToken.vy` inside the `contracts` folder. Then, add the following line:

  ```
  #pragma version ^0.3.0
  ```

This line means that you need at least Vyper version 3 to compile the code.

Then, add the following lines:

```
from vyper.interfaces import ERC20
from vyper.interfaces import ERC20Detailed

implements: ERC20
implements: ERC20Detailed
```

An interface is a list of function signatures and events without definitions. Vyper has built-in interfaces, which you can see here: `https://github.com/vyperlang/vyper/tree/master/vyper/builtins/interfaces`. At the time of writing, the following interfaces are available: ERC165, ERC20, ERC20Detailed, ERC4626, and ERC721.

In this example, you will have imported two interfaces: ERC20 and ERC20Detailed. You can take a look at how the ERC20 interface looks from the following URL: `https://github.com/vyperlang/vyper/blob/master/vyper/builtins/interfaces/IERC20.vyi`. Here's the content of the file:

```
# Events
event Transfer:
    sender: indexed(address)
    recipient: indexed(address)
    value: uint256

event Approval:
    owner: indexed(address)
    spender: indexed(address)
    value: uint256

# Functions
@view
@external
def totalSupply() -> uint256:
    ...
...
@external
def approve(_spender: address, _value: uint256) -> bool:
    ...
```

Basically, this is similar to the Solidity code of the ERC-20 standard that you've seen previously. Only this time, the code is in Vyper.

You can also take a look at the ERC20Detailed interface from the following URL: `https://github.com/vyperlang/vyper/blob/master/vyper/builtins/interfaces/IERC20Detailed.vyi`. Here's the content of the file:

```
@view
@external
def name() -> String[1]:
    ...

@view
@external
def symbol() -> String[1]:
    ...

@view
@external
def decimals() -> uint8:
    ...
```

This is like the optional code in Solidity, which you can implement while writing a smart contract. You don't have to implement these methods in order to make your smart contract ERC-20 compliant; however, these methods are quite handy.

There are optional methods other than the ERC-20 and ERC20Detailed standards, such as `ERC20Mintable`, `ERC20Burnable`, `ERC20Capped`, and `ERC20Pausable`:

- `ERC20Mintable` defines the method signatures with which to mint tokens (or add tokens to supply) in a smart contract
- `ERC20Burnable` defines the method signatures with which to burn tokens (or remove tokens from supply) in a smart contract
- `ERC20Capped` defines the method signatures with which to add an upper limit to a token supply
- `ERC20Pausable` defines the method signatures with which to add a pausing method to transferring and approving actions

Let's add lines of code to your smart contract:

```
implements: ERC20
implements: ERC20Detailed
```

These lines mean in order to compile a smart contract, you have to implement all method signatures and event signatures. Suppose you forgot to implement the `approve` method in the smart contract; you would receive the following error message if you compiled the smart contract:

```
ERROR: (VyperCompileError) HelloToken.vy
InterfaceViolation:Contract does not implement all interface functions
or events: approve
  contract "HelloToken.vy:6", line 6:0
       5
  ---> 6 implements: ERC20
  -------^
       7 implements: ERC20Detailed
```

You don't have to add those lines because you can still implement the methods of the ERC-20 standard or the optional `ERC20Detailed` standard. However, using the implements code forces you to implement all method signatures in the interfaces. Without this code, you might forget to implement one or two necessary important methods.

Now, let's add these lines of code:

```
event Transfer:
    sender: indexed(address)
    receiver: indexed(address)
    value: uint256

event Approval:
    owner: indexed(address)
    spender: indexed(address)
    value: uint256
```

The `Transfer` event will take place if the transfer action takes place. The `Approval` event will take place if the approval action takes place.

Now, let's add these state variables:

```
name: public(String[32])
symbol: public(String[32])
decimals: public(uint8)
```

By adding these state variables, you make sure your smart contract complies with the `ERC20Detailed` standard. However, notice that in the `ERC20Detailed` interface, `name`, `symbol`, and `decimals` are presented as method signatures, yet you have implemented state variables; this is fine because Vyper turns all public state variables into external view methods, meaning that it's redundant if you implement the following methods:

```
@view
@external
def name() -> String[1]:
    return self.name

@view
@external
def symbol() -> String[1]:
    return self.symbol

@view
@external

def decimals() -> uint8:
    return self.decimals
```

The `name` state variable refers to the name of the token, such as Uniswap or USD Coin. The `symbol` state variable refers to the symbol of the token, which is usually 3 or 4 characters, such as UNI or USDC. The `decimals` state variable refers to the decimals of the token. For example, 1 USDC has 6 decimals. So, the lowest amount of USDC that is bigger than 0 is 0.000001 USDC.

Let's add the following code to your smart contract:

```
balanceOf: public(HashMap[address, uint256])
allowance: public(HashMap[address, HashMap[address, uint256]])
totalSupply: public(uint256)
```

This case is the same as the previous one. In the interface, `balanceOf`, `allowance`, and `totalSupply` are presented as method signatures in the ERC20 interface. However, here, you have implemented them as state variables and that's fine because Vyper turns them into external methods.

The `balanceOf` state variable refers to the balance of an account or address. The `allowance` state variable refers to the allowance (from an address to another address) of how many tokens from the source address can be spent by the destination address. The `totalSupply` state variable refers to the total supply of the token in the smart contract.

Let's define the initialization method by adding the following code to the smart contract:

```
@external
def __init__(_name: String[32], _symbol: String[32], _decimals: uint8,
_supply: uint256):
    init_supply: uint256 = _supply * 10 ** convert(_decimals, uint256)
    self.name = _name
    self.symbol = _symbol
    self.decimals = _decimals
    self.balanceOf[msg.sender] = init_supply
    self.totalSupply = init_supply
    log Transfer(empty(address), msg.sender, init_supply)
```

In the initialization method, you set `name`, `symbol`, `decimals`, and `total supply` of the tokens from the arguments in the method. You also give the entire supply of the tokens to the deployer's address. Lastly, you log the transfer event.

You don't have to implement the initialization method this way. For example, you can give 50% of the total supply of tokens to your friend's address and another 50% to your sister's address; there's room for creativity here.

Now, let's add another method to the smart contract by adding the following code:

```
@external
def transfer(_to : address, _value : uint256) -> bool:
    self.balanceOf[msg.sender] -= _value
    self.balanceOf[_to] += _value
    log Transfer(msg.sender, _to, _value)
    return True
```

The method moves some tokens from the sender's account balance to another account balance. There is also a logging event for the transfer action. Remember that you don't have to implement the `transfer` method this way. You have room for creativity. For example, you can add a checking condition that only an account that has less than *X* number of tokens can transfer tokens. More than that, it needs permission from the owner. However, the core tenet of this method is to transfer tokens from the sender's address to another address.

Now, let's add another method by adding the following code:

```
@external
def transferFrom(_from : address, _to : address, _value : uint256) ->
bool:
    self.balanceOf[_from] -= _value
    self.balanceOf[_to] += _value
    self.allowance[_from][msg.sender] -= _value
```

```
    log Transfer(_from, _to, _value)
    return True
```

In this method, you transfer the tokens from one address to another address, but the twist is the source account is not the sender's address. Other than decreasing the number of tokens of the source address, this method also decreases the allowance from the source address to the sender address.

So, what's the purpose of the `transferFrom` method? Imagine a mother giving a certain allowance to her son. The son can transfer money from the mother's account to any destination up to a point. Perhaps the mother wants the son to act as an assistant and spend the money to buy groceries for the family.

Say the mother gives an allowance of as much as 1,000 "ABC" tokens to her son. Wouldn't it make sense for the son to transfer 1,000 ABC tokens from his mother's account to his account? That way, the mother cannot take it back again. The son can do this, but primarily, the `transferFrom` method is not created for that purpose. Usually, you do not give allowance to a person's account or an externally owned account; you give allowance to a smart contract. For example, you give allowance to an exchange smart contract, whereby you can sell your tokens for ETH. So, the smart contract needs the ability to transfer tokens from your balance to the buyer's balance in case there is a buyer for your tokens. This is where the `transferFrom` method comes in.

Now, let's add the `approve` method, whereby you can increase the allowance:

```
@external
def approve(_spender : address, _value : uint256) -> bool:
    self.allowance[msg.sender][_spender] = _value
    log Approval(msg.sender, _spender, _value)
    return True
```

In this method, you set the sender's allowance to the destination or spender address. The `Approval` event will be logged for this action. So, the `approve` method needs to be executed before you let other people execute the `transferFrom` method on your behalf.

That's it. You can compile the smart contract before deploying it to the blockchain. Then, you have a functional token smart contract. Refer to the previous chapters to learn how to compile and deploy smart contracts.

Suppose you want to add a method to increase the supply of your token; you're going to need to add the following method:

```
@external
def addSupply(_new_supply: uint256):
    assert msg.sender == self.owner
    self.balances[msg.sender] = _new_supply
```

This is perfectly fine. However, there is an optional standard for this method to increase the supply of tokens, and it's better to follow the standard. The standard is named `ERC20Mintable`. The method signature of adding the supply of tokens looks like this:

```
@external
def mint(_account: address, _amount: uint256) -> bool:
    ...
```

Then, it's up to you how you want to implement this method to increase the supply of tokens.

What if you want to decrease the supply of the token? You can name the method any name you like, but there is an optional standard for this method. The standard is named `ERC20Burnable`. The method's signature for decreasing the supply of tokens looks like this:

```
@external
def burn(_account: address, _amount: uint256) -> bool:
    ...
```

Now, you can implement the method any way you like, as long as the tokens can be decreased from an account by using this method.

So, if you want to add behaviors to a token smart contract, it's a good idea to check whether there are standards for the behaviors. That way, you can follow the method signatures; this increases interoperability.

Now that you've created a vanilla/generic ERC-20 smart contract, it's time to add some flavors to the smart contract. You can add custom behaviors to the ERC-20 token smart contract.

Selling tokens

Suppose you want to create an ERC-20 smart contract, and you want to create a method where people can buy your tokens with ETH; you can do this. There is no standard for this kind of method. So, you can name the method any name you like.

Let's say you want to sell your custom token for ETH. A crowdsourcing token is very easy to create on the Ethereum platform compared to the Bitcoin platform. You already know how to create a method in a smart contract to accept ETH. You also know how to increase the token balance of some accounts. To sell tokens, you must combine those two things. That's all.

This is the core of the **initial coin offering** (**ICO**) process. The currency of Ethereum is valuable. Although the price of ether fluctuates, 1 ether is valued at around USD 3,000 at the time of writing. People will pay real money for some ETH but not your custom token. To make your custom token worthy, you have to make it useful first, or at least make it appear useful. However, to do that, you need capital. So, why not sell some of your tokens (say 60%) to early adopters? They can then purchase your custom token with ETH. Then, you can withdraw ETH so that you can hire more programmers and rent an office to develop your new cryptocurrency. This is the basic idea. Of course, because an ICO involves a lot of money, it also attracts predators.

This is the crowd sale of token smart contracts; it is the same as your previous ERC-20 token smart contract's source code but with slight variations. Name this smart contract `CrowdSaleToken.vy` and save it inside the `contracts` directory. Refer to the code file in the following GitHub link for the full code: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_12/erc20/contracts/CrowdSaleToken.vy`.

Let's discuss this smart contract line by line. Here are the first lines of the smart contract:

```
#pragma version ^0.3.0

from vyper.interfaces import ERC20
from vyper.interfaces import ERC20Detailed

implements: ERC20
implements: ERC20Detailed
```

You've seen these lines in the vanilla ERC-20 smart contract. So, let's see the next lines:

```
event Transfer:
    sender: indexed(address)
    receiver: indexed(address)
    value: uint256

event Approval:
    owner: indexed(address)
    spender: indexed(address)
    value: uint256

event Payment:
    buyer: indexed(address)
    value: uint256
```

The `Transfer` and `Approval` events are the standard ERC-20 events that you must implement. The `Payment` event is the custom event that will be logged when a buyer purchases tokens from this smart contract.

Here are the next lines:

```
name: public(String[32])
symbol: public(String[32])
decimals: public(uint8)

balanceOf: public(HashMap[address, uint256])
allowance: public(HashMap[address, HashMap[address, uint256]])
totalSupply: public(uint256)
```

These are the standard ERC-20 state variables, which you've seen in the previous smart contract.

Then, you should add custom state variables by adding the following code:

```
ethBalances: public(HashMap[address, uint256])

beneficiary: public(address)
minFundingGoal: public(uint256)
maxFundingGoal: public(uint256)
amountRaised: public(uint256)
deadline: public(uint256)
price: public(uint256)
fundingGoalReached: public(bool)
crowdsaleClosed: public(bool)
```

The `ethBalances` state variable is a mapping variable that tracks how much ETH a buyer has stored in a smart contract. The `beneficiary` state variable refers to the owner of the smart contract who will get ETH from buyers; in other words, it's a seller. The `minFundingGoal` state variable refers to the minimum ETH you want to get for this smart contract. The `maxFundingGoal` state variable refers to the maximum ETH you want to get for selling tokens. This means you will prevent buyers from buying tokens when you have reached this goal. The `amountRaised` state variable refers to the amount of ETH you have received from selling tokens. The `deadline` state variable refers to the latest date buyers can buy tokens. The `price` state variable refers to the price of tokens in ETH. The `fundingGoalReached` state variable refers to whether you have reached the minimum funding goal or not. The `crowsaleClosed` state variable refers to whether the sale of tokens has been stopped or not.

Next, we have the initialization method:

```
@external
def __init__(_name: String[32], _symbol: String[32], _decimals: uint8,
_supply: uint256):
    init_supply: uint256 = _supply * 10 ** convert(_decimals, uint256)
    self.name = _name
    self.symbol = _symbol
    self.decimals = _decimals
    self.balanceOf[msg.sender] = init_supply
    self.totalSupply = init_supply
    log Transfer(empty(address), msg.sender, init_supply)

    self.beneficiary = msg.sender
    self.minFundingGoal = as_wei_value(30, "ether")
    self.maxFundingGoal = as_wei_value(50, "ether")
    self.deadline = block.timestamp + 3600 * 24 * 100 # 100 days
    self.price = as_wei_value(1, "ether") / 100
```

```
    self.fundingGoalReached = False
    self.crowdsaleClosed = False
```

First, you set up the standard properties of the ERC-20 smart contract. The total coin supply in this smart contract is defined by the argument, as in the previous ERC-20 smart contract.

You want to raise at least 30 ether and up to a maximum of 50 ether.

The deadline is set to 100 days from the time when the smart contract is deployed on the blockchain. `block.timestamp` is approximately the current time or the time when the block containing this smart contract code is confirmed.

The price of one coin is set to 0.01 ether. This means that 1 ether can buy 100 coins of your token.

Now, let's see the method used to sell tokens and receive ETH:

```
@external
@payable
def __default__():
    assert msg.sender != self.beneficiary
    assert self.crowdsaleClosed == False
    assert self.amountRaised + msg.value < self.maxFundingGoal
    assert msg.value >= as_wei_value(0.01, "ether")
    self.ethBalances[msg.sender] += msg.value
    self.amountRaised += msg.value
    tokenAmount: uint256 = msg.value / self.price
    self.balanceOf[msg.sender] += tokenAmount
    self.balanceOf[self.beneficiary] -= tokenAmount
    log Payment(msg.sender, msg.value)
```

This is the method that the user can use to buy the token. `__default__` is a default fallback function. If someone does not execute a method on the smart contract and pays ether, this function will be executed. Actually, you don't have to use the default function to accept the payment; you can use an ordinary method, just like you did in the previous smart contracts.

In this payment method, you ensure that the buyer is not the beneficiary, that the crowd sale is still happening, and that the amount that has been raised with the ether sent to this method does not exceed the maximum funding goal of 50 ether. Lastly, each purchasing action must be at least 0.01 ether. Then, you increase the balance of ether for this buyer as well as increase the amount of ether raised. We then check the number of coins they purchased by dividing the amount of ether by the price of one coin.

Finally, you have to increase the balance of the token for this buyer and decrease the balance of the token for the owner of the smart contract. You also should not forget to log this event.

Now, let's have a look at the method to check whether you've reached the funding goal or not by looking at the next code lines:

```
@external
def checkGoalReached():
    assert block.timestamp > self.deadline
    if self.amountRaised >= self.minFundingGoal:
        self.fundingGoalReached = True
    self.crowdsaleClosed = True
```

First, we make sure that this method can only be executed successfully if the deadline has passed. If the amount raised is more than the minimum funding goal, we set the `fundingGoalReached` variable to `True`. Then, finally, we set the `crowdsaleClosed` variable to `True`.

For the sake of simplicity, you only check whether the `block.timestamp` variable is greater than the deadline. However, the timestamp in the block could be filled with anything the miner likes; it does not have to be the current time that the block is confirmed. However, of course, if the miner gave the past timestamp as the value for `block.timestamp`, all other miners would reject it. Similarly, if the miner gives a future timestamp that is really far away (for example, a year ahead), as the value for `block.timestamp`, all other miners would also reject it. To make the checking of the deadline process more secure, you have to combine it with `block.number` to check how many blocks have been confirmed since this smart contract launched.

The last custom method for this smart contract is to withdraw ETH:

```
@external
def safeWithdrawal():
    assert self.crowdsaleClosed == True
    if self.fundingGoalReached == False:
        if msg.sender != self.beneficiary:
            if self.ethBalances[msg.sender] > 0:
                ethBalance: uint256 = self.ethBalances[msg.sender]
                self.ethBalances[msg.sender] = 0
                self.balanceOf[self.beneficiary] += self.
balanceOf[msg.sender]
                self.balanceOf[msg.sender] = 0
                send(msg.sender, ethBalance)
    if self.fundingGoalReached == True:
        if msg.sender == self.beneficiary:
            if self.balance > 0:
                send(msg.sender, self.balance)
```

The `safeWithdrawal` method runs differently, depending on whether the funding goal is attained or not. Inside the preceding method, you have to make sure that the crowd sale has already closed. If the funding goal is not reached, you make sure that every buyer can get their ether back. If the funding goal is reached, then we make sure that the beneficiary can withdraw all the ether in the smart contract. The remainder of the methods are the same as in the previous smart contract. However, we add a number of assertions to make sure that these methods can only be executed after the crowd sale has been closed.

Then, the rest of the requirements are the standard methods of ERC-20 smart contracts, such as `transfer`, `transferFrom`, and `approve`.

As you can see, you've modified the vanilla ERC-20 smart contract so that it has the utility to sell tokens. ERC-20 smart contracts don't have to be vanilla; you can add many innovations. However, because your smart contract follows the ERC-20 standard, people and other smart contracts can interact with many ERC-20 smart contracts easily.

After creating the crowd sale smart contract, you can compile and release it to the blockchain. Hopefully, people will buy your tokens!

But don't forget to secure the smart contract!

Security aspects

The ERC-20 smart contracts deal with values or money. Of course, if tokens were not worthwhile, then we wouldn't be having this conversation. However, there are ERC-20 smart contracts that are valuable. For example, USDC or USDT are ERC-20 smart contracts. They are valuable because 1 USDT or USDC can be redeemed with 1 US Dollar.

So, it would be a nightmare if people could increase their balances in USDT or USDC smart contracts illegally.

You want to secure the ERC-20 smart contracts that you're going to create. There are some steps you can take to increase the security of the ERC-20 smart contracts.

First, you should follow the ERC-20 standard as closely as possible. Important functions such as `transfer`, `transferFrom`, and `approve` have been written and audited by other people. So you can just use them and not write them from scratch. The functions have been battle-tested. You can add other improvements to the smart contract while maintaining the important functions written by other people.

Let's take a look at one security issue in your ERC-20 smart contract to see what kind of problems you might be encountering. Here's an implementation of the `approve` method:

```
@external
def approve(_spender : address, _value : uint256) -> bool:
    self.allowance[msg.sender][_spender] = _value
    log Approval(msg.sender, _spender, _value)
    return True
```

Imagine this situation. Mary wants to delegate some financial decisions to her friend, Jack. She broadcasts the `approve` method transaction to the blockchain. She approves of Jack spending `100` ABC tokens. All is well. Then, Mary changes her mind. She decides that Jack is not to be trusted to spend `100` ABC tokens; she wants to lower the allowance to `12` ABC tokens. So, she broadcasts the `approve` method transaction to the blockchain, but this time, the argument is `Jack` in the first argument and `12` in the second argument. However, Jack realizes this change in transaction capability and decides to front-run it. Jack broadcasts the `transferFrom` transaction so he can transfer `100` ABC tokens on Mary's behalf to his address before the new allowance sets in. Then, after the second `approve` method transaction is confirmed, Jack can spend another `12` ABC tokens.

What Mary should do is broadcast the `approve` method transaction with a `0` value first. That way, if Mary can see that Jack is front-running the transaction, Mary can decide not to allow Jack to spend `12` ABC tokens again. Mary can limit the damage to `100` ABC tokens.

However, as the token author, you can write some mitigations to the `approve` and `transferFrom` methods so that Mary doesn't have to worry about this. Instead of setting a new value in the `approve` method, you can use increasing or decreasing amounts. Instead of setting `12` as the new value, you can decrease the `88` value. So, if Jack front-runs Mary's transaction, you, as the smart contract's author, can set it to `0` because the existing value is already `0`. No need to decrease it again.

You can also implement a fail-safe mode. Say there is a breach in your smart contract. You, as the owner of the smart contract, can pause all the operations in the smart contract. That way, you can take time to figure out how to mitigate the damage on your smart contract. As discussed previously, the ERC20Pausable standard can be used to implement this. If the smart contract is paused, nobody can transfer their tokens.

When you want to investigate the issue within your smart contract, you need data and logs. So, any event that alters the state of your smart contract needs to be logged. If you change the owner of the smart contract, you log the event. If you mint new tokens, you log the event. If you burn new tokens, you log the event. Always create events and log them for important methods.

You can also analyze your smart contract with a security tool. One example of this is Slither (`https://github.com/crytic/slither`). However, Slither's support for Vyper is not really great (at the moment of writing). However, if you write smart contracts in Solidity, it's a very helpful tool. It can analyze your smart contracts and generate reports.

If you write smart contracts in Solidity, you can also use tools and libraries from OpenZeppelin (`https://www.openzeppelin.com/contracts`). Since Solidity supports inheritance, you can make your smart contracts inherit classes from Solidity and then add your custom methods.

Summary

In this chapter, you have learned how to create tokens on top of Ethereum. First, you created a very simple token smart contract and learned why the token smart contract offers something valuable that is missing from traditional applications. Then, you created an ERC-20 standard token smart contract by implementing the methods required by the ERC-20 standard. You learned what the ERC-20 standard is and why it's needed. Then, you customized an ERC-20 smart contract by adding a method to sell tokens for ETH. Finally, you studied how to secure an ERC-20 smart contract. In the next chapter, you'll learn how to create an NFT smart contract.

13
How to Create an NFT

In this chapter, you're going to learn about **non-fungible token** (**NFT**) technology. After the ERC-20 token technology became mainstream on Ethereum, NFTs also became more popular and are being used by many people. People buy and sell digital art as NFTs. Some NFTs, such as CryptoPunks and Bored Ape Yacht Club, have become so popular and valuable that the average price of a single item of CryptoPunks is 40-60 ETH or $140,000 to $290,000 as of this writing takes place. With the knowledge from this chapter, you can launch an NFT collection similar to Bored Ape Yacht Club!

This chapter will cover the following topics:

- What is an NFT?
- ERC-721 standard
- Creating the NFT smart contract

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_13`.

What is an NFT?

As mentioned, NFT is a short form of non-fungible token. The definition of "token" is the same as the definition of "token" in the ERC-20 token. A token is a representation of "something." For example, ERC-20 tokens can represent digital currency or virtual money. As for NFT, there is "non-fungible" in NFT.

So what does "fungible" mean? When something is fungible, it means one token is as worthy as another token. For example, say you created 1000 ERC-20 tokens named XYZ. This means that any one particular XYZ token is as valuable as any other XYZ token.

If that confuses you, you can swap XYZ with US dollars. Let's assume that I have USD 100 in my wallet and you have USD 100 in your wallet. My USD 100 is as valuable as your USD 100. In other words, USD is fungible. 1 USD has the same value as the other 1 USD.

Coming back to the NFT, the token in an NFT is non-fungible. 1 NFT token is not as valuable as 1 other NFT token. Every NFT token is unique. Suppose you have an NFT collection that consists of 10 NFT tokens. There are NFT token number 1, NFT token number 2, NFT token number 3, and so on, until NFT token number 10. The NFT token number 1 is different from the NFT token number 2. Every NFT token is unique. You cannot treat the NFT token number 3 as you treat the NFT token number 7.

If this is too abstract, you can switch an NFT collection with a Pokemon card collection. Again, let's assume that our Pokemon card collection has 10 items. Item number 1 is a card with a picture of Pikachu (a beloved Pokemon character). Item number 2 is a card with the Bulbasaur (another beloved Pokemon character) picture. You cannot say that the Pikachu card is as valuable as the Bulbasaur card. They are different. They have different values. A Pikachu card could be more valuable than a Bulbasaur card or vice versa.

If ERC-20 tokens are similar to US dollars, then NFT tokens are similar to Pokemon cards. In other words, if ERC-20 tokens are virtual currency, then NFT tokens are digital art.

> **Note**
>
> However, digital arts in this context are not the same as digital pictures that a person creates using an application such as Photoshop or Adobe Illustrator. A digital picture is a picture that you must see on a digital platform such as mobile phones, laptops, or PCs. A Pokemon card in the form of an NFT has something that a digital picture does not have, which is embedded ownership. Also, NFT gives you an option to put the data into the blockchain.

Coming to the ownership value proposition, every NFT comes with the owner. In the context of digital pictures, it's different. A digital picture of a cat, for example, doesn't imply who the owner is. You have to put the ownership of the digital cat picture somewhere else. For example, you can put it in the database of a web application that keeps images, including the digital cat picture. The ownership can be put inside a table with a column named `url_of_picture`, referring to the location of the digital cat picture, and a column named `profile_id`, referring to the ID of a row in the `profiles` table, which includes your information.

We can thus say that an NFT token comes with a sense of ownership. It's the same as in the case where every ERC-20 token comes with the ownership information. The information is in the `balanceOf` state variable in the smart contract.

```
balanceOf[address] = amount of tokens
```

The NFT use case is also not limited to digital art on the blockchain. It can refer to other things such as concert tickets. A ticket can be tied to a seat. A seat in the front row is different from a seat in the second row although they are priced the same. So, in a way, a ticket that is tied to a seat is unique or non-fungible. It's like an NFT.

Enough of the theory. Let's create a simple NFT smart contract to better illustrate the idea. Let's install the Ape Framework and create a project:

```
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install eth-ape'[recommended-plugins]'==0.7.23
(.venv) $ mkdir nft
(.venv) $ cd nft
```

Then, initialize the project:

```
(.venv) $ ape init
Please enter project name: HelloNFT
SUCCESS: HelloNFT is written in ape-config.yaml
```

After this, create a new file named `VerySimpleNFT` inside the `contracts` folder and then add the following code to it:

```
#pragma version ^0.3.0

ownerOf: public(HashMap[uint256, address])

@external
def __init__():
    for i in range(10):
        self.ownerOf[i] = msg.sender

@external
def transfer(tokenId: uint256, destination: address):
    if self.ownerOf[tokenId] == msg.sender:
        self.ownerOf[tokenId] = destination
```

To track the ownership of an NFT, you use the `ownerOf` state variable. In the ERC-20 token, you use the `balanceOf` state variable to keep track of how many tokens an address has. Here, you use the `ownerOf` state variable to keep track of which NFT an account has. In the `balanceOf` variable, the key to the variable is the address and the value of the variable is a number. The variable answers the question a user has about how many tokens there are. Notice that you don't need to track which tokens an address has because every token in the ERC-20 smart contract has the same value or is fungible.

However, in the NFT smart contract, you need to track the ownership. So, the key of the `ownerOf` state variable is an integer and the value of the `ownerOf` state variable is an address. The variable answers the question of who owns the NFT. An NFT in the smart contract is represented by a number or an integer. NFT 0 belongs to address A. NFT 1 belongs to address B, and so on. NFT 0 is different from NFT 1.

To transfer an NFT or the ownership of an NFT, you can use the `transfer` method. In this method, you change the value of the `ownerOf` state variable with a certain address key. If NFT 0 belongs to address A, address A can change the owner of NFT 0 to another address such as address B. So, in the end, NFT 0 belongs to address B.

That's the core of an NFT smart contract.

Let's create the test of this smart contract. Create `conftest.py` inside the `tests` folder and add the following code to it:

```
import pytest


@pytest.fixture
def deployer(accounts):
    return accounts[0]

@pytest.fixture
def contract(deployer, project):
    return deployer.deploy(project.VerySimpleNFT)
```

This is the test helper file so you can get the deployed smart contract and the deployer's address. Then, let's create the test file named `test_VerySimpleNFT.py` inside the `tests` folder and add the following code to it:

```
def test_ownerOf(contract, deployer):
    for i in range(10):
        assert deployer == contract.ownerOf(i)

def test_transfer(contract, deployer, accounts):
    destination_account = accounts[1]
    tokenId = 2
    contract.transfer(tokenId, destination_account, sender=deployer)
    assert destination_account == contract.ownerOf(tokenId)
    for i in range(2):
        assert deployer == contract.ownerOf(i)
    for i in range(3, 10):
        assert deployer == contract.ownerOf(i)
```

In the `test_ownerOf` function, you test that the owner of all NFT tokens is the deployer. Then in the `test_transfer` function, to test whether the transfer function works or not, you call the transfer function first, then check that the owner of the NFT with an ID of 2 changed from the deployer's address to the destination address.

You can compile the smart contract by using this command:

```
(.venv) $ ape compile
```

Then, you can run the test by using this command:

```
(.venv) $ py.test tests/test_VerySimpleNFT.py
```

But how do you store digital art such as a picture on the smart contract? It's basically a convention. NFT with an ID of `0` refers to the Pikachu card. An NFT with an ID of `1` refers to Bulbasaur. It's up to you how you set up the convention. Later, you'll see how to place the convention inside the smart contract.

So, why is it a big deal?

In the ERC-20 tokens case, you appreciate the value of the transparency of the rules of ERC-20 smart contracts and the censorship resistance of ERC-20 tokens. When someone creates a smart contract where they cannot mint or create additional tokens, it means they cannot change the rules on the fly. People can see that and decide whether they want to jump into using the smart contract or not. In a certain smart contract, if a sender transfers a number of tokens to another user, and the rule of the smart contract enforces it, then the sender cannot undo the transaction, no matter how desperate the sender is. A deployer of the smart contract can create a smart contract where even they cannot censor the transactions happening inside the smart contract.

In the NFT case, it's similar. You can deploy a smart contract where any user cannot undo the transfer of an NFT, not even yourself. You can create a limited amount of NFT tokens in a smart contract. For example, you can create a collection of 10 tokens representing Pokemon cards inside a smart contract and a rule where you cannot add additional tokens after the smart contract is deployed. So, imagine, you buy NFT token 1 from the deployer of the NFT smart contract or perhaps you get NFT token 1 as a gift from the deployer. Then, someday, your NFT might become very valuable. When this happens, the deployer could end up regretting their decision. They want to get it back but they cannot do that. So, this means that your NFT belongs to you, and no one can take it away from you.

In this way, the NFT technology brings digital ownership to rare items. The ownership can be proven right away on the blockchain. It can thus be said that if the ERC-20 tokens technology brings money to blockchain, the NFT technology brings digital arts to blockchain.

Another interesting thing about NFTs is that you can see the history of ownership of a specific NFT. People can see whether you get the NFT from the primary sales (you buy the NFT from the owner of the smart contract) or from the secondary sales (you buy from a person who bought it from another person). People can verify that you own the NFT from the start and never let go of it, thereby proving your loyalty.

The use case of it is that an artist can sell NFTs to their fans and you, as the loyal fan, can prove that you have owned the NFT from the beginning. The artist can see the blockchain to verify you're a loyal one. The artist can reward you with some gifts.

In those ways, NFTs can be a gateway or a filter to other things such as an exclusive Discord community. If you own a certain NFT, you can join the Discord community. Otherwise, you cannot.

An interesting thing is you can transfer or sell your NFT to somebody else. It unlocks another use case. Suppose you buy an NFT to subscribe to a movie streaming service such as Netflix or Disney+. If you own the NFT, you can watch the movies from the service. If you don't, you can't watch them. But remember that you can transfer or sell the NFT to someone else. It means you can transfer or sell your movie streaming subscription!

NFT can also represent assets used in video games, such as characters, virtual lands, weapons, or clothes. Usually, people can buy upgrades in video games they play through in-app purchases. However, this thing can also be done with NFTs.

Aside from representing virtual items, NFTs can represent real-world assets, such as real estate. Owning an NFT can grant you usage of houses or properties.

NFTs can also be used in education. For example, your education certificates can be represented by NFTs. To prove whether you've undergone education in a certain university, you can give an NFT that is granted by the university to you as a credential.

Although you can write an NFT smart contract freely, there is a standard that you can use.

ERC-721 standard

Just like tokens on Ethereum have the ERC-20 standard, NFTs on Ethereum have the ERC-721 standard. To make a smart contract ERC-721 compliant, a smart contract needs to fulfill the ERC-721 standard and the ERC-165 standard. This standard is useful if you want your NFT smart contract to be interoperable with other smart contracts, for example, if you want other people to be able to trade NFT tokens from your NFT smart contract.

To fulfill the ERC-721 standard, you need to implement these events and function signatures:

```
event Transfer(address indexed _from, address indexed _to, uint256
indexed _tokenId);
event Approval(address indexed _owner, address indexed _approved,
uint256 indexed _tokenId);
event ApprovalForAll(address indexed _owner, address indexed _
operator, bool _approved);
function balanceOf(address _owner) external view returns (uint256);
function ownerOf(uint256 _tokenId) external view returns (address);
function safeTransferFrom(address _from, address _to, uint256 _
tokenId, bytes data) external payable;
function safeTransferFrom(address _from, address _to, uint256 _
```

```
tokenId) external payable;
function transferFrom(address _from, address _to, uint256 _tokenId)
external payable;
function approve(address _approved, uint256 _tokenId) external
payable;
function setApprovalForAll(address _operator, bool _approved)
external;
function getApproved(uint256 _tokenId) external view returns
(address);
function isApprovedForAll(address _owner, address _operator) external
view returns (bool);
```

The preceding code is written in Solidity syntax, and you need to implement the functions and events in Vyper. Let's discuss the events and function signatures one by one.

Here is the explanation of the functions and the events:

- The first event you need to implement is the `Transfer` event. This event is triggered when a user transfers an NFT to another user.
- The second event you need to implement is the `Approval` event. This event is triggered when an owner allows another user to transfer an NFT on behalf of the owner. This approval is only for one NFT.
- The last event you need to implement is the `ApprovalForAll` event. This event is triggered when an owner allows another user to transfer all NFT tokens on behalf of the owner. This approval is for all NFT tokens owned by the owner.
- The first function you need to implement is the `balanceOf` function. This function's purpose is to tell people how many NFT tokens a user has in this smart contract.
- The second function you need to implement is the `ownerOf` function. This function is to tell people who is the owner of a specific NFT.
- The third and fourth functions you need to implement are the `safeTransferFrom` functions. They share the name of the function, but the arguments are different. The first `safeTransferFrom` function has the `data` argument with the `bytes` data type. The function transfers an NFT from a sender to a destination. But what is the data argument for? If you transfer to a normal address, the data argument will not be used. The data argument is used only when the destination is a smart contract. When the destination is the smart contract, the `safeTransferFrom` function will call the `onERC721Received` function on the smart contract with the data argument.
- The fifth function you need to implement is the `transferFrom` function. The function's purpose is the same as the `safeTransferFrom` function's purpose. The only difference is that the `transferFrom` function doesn't care whether the destination is a smart contract or not.

- The sixth function you need to implement is the `approve` method. This is the function to approve a specific NTF so a user can transfer it on behalf of another user.
- The seventh function you need to implement is the `setApprovalForAll` function. Rather than approving one NFT only, this function approves all NFT tokens so a user can transfer all of them on behalf of another user.
- The eighth function you need to implement is the `getApproved` function. This is the function to check whether a user can transfer a specific NFT on behalf of another user.
- The ninth function you need to implement is the `isApprovedForAll` function. This is the function to check whether a user can transfer all NFT tokens on behalf of another user.

To fulfill the ERC-165 standard, you need to implement this function signature:

```
function supportsInterface(bytes4 interfaceID) external view returns
(bool)
```

The `supportsInterface` function's purpose is to tell people whether the smart contract supports an interface or not. For example, if you want to know whether a smart contract supports the ERC-721 interface or, in other words, whether a smart contract is an NFT smart contract or not, you can call this method.

As in the case of the ERC-20 standard, the ERC-721 standard has optional function signatures you can implement, which are listed as follows:

```
function name() external view returns (string _name);
function symbol() external view returns (string _symbol);
function tokenURI(uint256 _tokenId) external view returns (string);
```

Let's look into what these function signatures can do:

- The `name` function returns the name of the NFT smart contract, for example, `"Pokemon Collection"`.
- The `symbol` function returns the symbol of the NFT smart contract, for example, `"POK"`.
- The `tokenURI` function returns the URI of the NFT token. This is the place where you can view the NFT itself.

Does it look like the URL of the picture of the NFT? Not quite. The file type is not a picture file type, but a JSON file type. The following example of a JSON file type is from a Pudgy Penguin NFT:

```
{
  "attributes": [
    {
      "trait_type": "Background",
      "value": "Blue"
```

```
    },
    {
      "trait_type": "Skin",
      "value": "Olive Green"
    },
    {
      "trait_type": "Body",
      "value": "Shirt Red"
    },
    {
      "trait_type": "Face",
      "value": "Normal"
    },
    {
      "trait_type": "Head",
      "value": "Durag Red"
    }
  ],
  "description": "A collection 8888 Cute Chubby Pudgy Penquins sliding
around on the freezing ETH blockchain.",
  "image": "ipfs://QmNf1UsmdGaMbpatQ6toXSkzDpizaGmC9zfunCyoz1enD5/
penguin/3.png",
  "name": "Pudgy Penguin #3"
}
```

The image of the NFT token can be accessed in the image field, which is `ipfs://QmNf1UsmdGaMbpatQ6toXSkzDpizaGmC9zfunCyoz1enD5/penguin/3.png`. To access IPFS from the browser, you can open this URL: `https://cloudflare-ipfs.com/ipfs/QmNf1UsmdGaMbpatQ6toXSkzDpizaGmC9zfunCyoz1enD5/penguin/3.png`. If you open the image file, you can view a cute penguin with a red shirt.

At this stage, you may be wondering: why don't we put an image URL in the `tokenURI` function? Because an image is not sufficient to explain the NFT. A JSON file can describe the NFT in more detail. You can see the traits of this NFT, such as the `"Head"` trait type has the `"Durag Red"` value. The attributes determine the rarity of an NFT. Some trait types are rarer than other trait types. This combination of attributes is suitable to make an NFT smart contract similar to a card collection.

However, an NFT doesn't have to be represented as an image. It could be represented as a ZIP file or a PDF file. You just have to put the location of the ZIP file or the PDF file in the JSON file.

There is an optional standard that you can choose to implement or not, which is called **ERC721Enumerable**. You need to implement the following function signatures:

```
function totalSupply() external view returns (uint256);
function tokenByIndex(uint256 _index) external view returns (uint256);
function tokenOfOwnerByIndex(address _owner, uint256 _index) external
view returns (uint256);
```

The `totalSupply` function tells people how many NFT tokens a smart contract has.

The `tokenByIndex` function is used to get the token ID by an index. Suppose you have 10 NFT tokens. The indexes are `0`, `1`, and so on, until `9`. However, the token ID does not have to follow that sequence. The token ID can be `99`, `100`, `88`, `111`, `333`, `43`, `44`, `45`, `46`, and `47`. So, index `0` gets a token ID of `99`, index `1` gets a token ID of `100`, and so on.

The `tokenByOwnerOfIndex` function gets the token ID from an index owned by a specific user.

This optional standard is useful if you want your NFT tokens to be enumerated or iterated.

Now that we have understood and familiarized ourselves with the ERC-721 standard, let's create the smart contract!

Creating the NFT smart contract

Let's create an NFT smart contract named `HelloNFT`. You can reuse the project directory you've created.

The smart contract code is quite long. You can get the smart contract from this URL: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_13/nft/contracts/HelloNFT.vy`. The code looks like this:

```
#pragma version ^0.3.0

# Adapted from https://github.com/vyperlang/vyper/blob/master/
examples/tokens/ERC721.vy

from vyper.interfaces import ERC165
from vyper.interfaces import ERC721

implements: ERC721
implements: ERC165

event Transfer:
    _from: indexed(address)
    _to: indexed(address)
    _tokenId: indexed(uint256)
```

```
...
@view
@external
def tokenURI(tokenId: uint256) -> String[132]:
    return concat(self.baseURL, uint2str(tokenId))
```

You can download the smart contract file and put it inside the `contracts` folder. You can also create a new file named `HelloNFT.vy` inside the `contracts` folder and then add code line by line.

Interfaces, events, and state variables

You start by adding the following code:

```
#pragma version ^0.3.0

# Adapted from https://github.com/vyperlang/vyper/blob/master/
examples/tokens/ERC721.vy

from vyper.interfaces import ERC165
from vyper.interfaces import ERC721

implements: ERC721
implements: ERC165
```

The pragma line says that you need at least Vyper version 0.3.x to compile the smart contract file. Then, there is a comment line telling people that this smart contract is adapted from another smart contract from the official Vyper GitHub repository. After that, you imported the ERC165 and ERC721 interfaces before calling `implements` on them. This way, you force your smart contract to be ERC-721 compliant.

Then, you add the required events for the ERC-721 standard by adding the following code:

```
event Transfer:
    _from: indexed(address)
    _to: indexed(address)
    _tokenId: indexed(uint256)

event Approval:
    _owner: indexed(address)
    _approved: indexed(address)
    _tokenId: indexed(uint256)

event ApprovalForAll:
    _owner: indexed(address)
    _operator: indexed(address)
    _approved: bool
```

The `Transfer` event in the previous code will be used when someone transfers an NFT to other people. The `Approval` event will be used when an owner of an NFT approves another person so that person can transfer the NFT on behalf of the owner. `ApprovalForAll` will be used when an owner of NFT tokens approves another person so that person can transfer all NFT tokens on behalf of the owner.

Then, you need to write the `ERC721Receiver` interface by adding the following code:

```
interface ERC721Receiver:
    def onERC721Received(
            _operator: address,
            _from: address,
            _tokenId: uint256,
            _data: Bytes[1024]
        ) -> bytes4: nonpayable
```

This interface enables you to execute the `onERC721Received` method of a smart contract if you transfer an NFT to a smart contract.

It would be nice if your NFT smart contract had a name and a symbol so people can differentiate it from other NFT smart contracts. Add the `name` and `symbol` state variables:

```
name: public(String[32])

symbol: public(String[32])
```

The next code you're going to add is the `HashMap` variables, which track the NFT, owners, and approvals:

```
idToOwner: HashMap[uint256, address]

idToApprovals: HashMap[uint256, address]

ownerToNFTokenCount: HashMap[address, uint256]

ownerToOperators: HashMap[address, HashMap[address, bool]]
```

The `idToOwner` state variable links the NFT token ID to the owner's address. The `idToApprovals` state variable links the NFT token ID to the account address that can transfer this NFT on behalf of the owner. The `ownerToNFTokenCount` state variable counts how many NFT tokens the owner has. The `ownerToOperators` state variable links the owner to the account addresses that can transfer the owner's NFT tokens on behalf of the owner.

Then, you need to add the following code to track the minter and the base URL:

```
minter: address

baseURL: String[53]
```

The `minter` state variable is the account address that can mint or create NFT tokens. The `baseURL` state variable is the base URL, which is part of the URL of the NFT assets.

Then, you need to create constant values for the interfaces that this smart contract supports by adding the following code:

```
SUPPORTED_INTERFACES: constant(bytes4[2]) = [
    # ERC165 interface ID of ERC165
    0x01ffc9a7,
    # ERC165 interface ID of ERC721
    0x80ac58cd,
]
```

The smart contract only supports the ERC165 interface and the ERC721 interface. The byte value comes from the `keccak256` function with the method name in the interface as the argument. So, this is how we get the bytes.

First, get the function selector with the `keccak256` function of a method, for example, the `balanceOf` function:

```
keccak256("balanceOf(address)") =
0x70a08231c33daf83428b320564fe444fb3d38c5bc90914637c72f46384995e5f
```

Then, you get just 4 bytes of this hash: `0x70a08231`. Do that for all methods in the ERC721 interface. Then, you will carry out an XOR operation on all of the results.

Implementing functions

Now, it's time to create the initialization function:

```
@external
def __init__():
    self.minter = msg.sender
    self.baseURL = "https://packtpub.com/metadata/"
    self.name = "Hello NFT"
    self.symbol = "HEL"
```

You initialized the `minter` variable, the `baseURL` variable, the `name` variable, and the `symbol` variable. Your NFT token name will be `"Hello NFT"`. You will store the NFT assets inside `https://packtpub.com/metadata`. Yes, you don't have to store NFT assets on IPFS. You can store them on ordinary web servers.

Then, you add the `supportsInterface` method:

```
@view
@external
def supportsInterface(interface_id: bytes4) -> bool:
    return interface_id in SUPPORTED_INTERFACES
```

Your NFT smart contract only supports values defined in `SUPPORTED_INTERFACES`, which are the ERC-721 interface and the ERC-165 interface. Users can call this method to check whether their smart contract is ERC-721 compliant or not.

Now, let's implement the `balanceOf` function by adding the following code:

```
@view
@external
def balanceOf(_owner: address) -> uint256:
    assert _owner != empty(address)
    return self.ownerToNFTokenCount[_owner]
```

This function's purpose is to check how many NFT tokens an owner has. You can get it from the `ownerToNFToken` state variable. You need to make sure the address is not the empty address because the empty address cannot own NFT tokens. When you transfer an NFT to the empty address, it means you destroy the NFT.

Now, let's implement the `ownerOf` function by adding the following code:

```
@view
@external
def ownerOf(_tokenId: uint256) -> address:
    owner: address = self.idToOwner[_tokenId]
    assert owner != empty(address)
    return owner
```

The function of the `ownerOf` function is to find the owner from an NFT token ID. You can get it from the `idToOwner` state variable. Here, as well, you need to make sure the address is not the empty address.

Now, let's implement the `getApproved` function by adding the following code:

```
@view
@external
def getApproved(_tokenId: uint256) -> address:
    assert self.idToOwner[_tokenId] != empty(address)
    return self.idToApprovals[_tokenId]
```

The `getApproved` function's goal is to get the approval address of an NFT. This approval address can transfer the NFT on behalf of the owner. You get it from the `idToApprovals` variable. Make sure the NFT token ID has an owner first.

Now, let's implement the `isApprovedForAll` function by adding the following code:

```
@view
@external
def isApprovedForAll(_owner: address, _operator: address) -> bool:
    return (self.ownerToOperators[_owner])[_operator]
```

The function of the `isApprovedForAll` function is to check whether an address has the approval to transfer all NFT tokens from an owner on behalf of the owner. You get it from the `ownerToOperators` variable.

Next, you implement a helper function to check whether an address can transfer an NFT on behalf of the owner by adding the following code:

```
@view
@internal
def _isApprovedOrOwner(_spender: address, _tokenId: uint256) -> bool:
    owner: address = self.idToOwner[_tokenId]
    spenderIsOwner: bool = owner == _spender
    spenderIsApproved: bool = _spender == self.idToApprovals[_tokenId]
    spenderIsApprovedForAll: bool = (self.ownerToOperators[owner])
[_spender]
    return (spenderIsOwner or spenderIsApproved) or
spenderIsApprovedForAll
```

To be able to transfer an NFT on behalf of the owner, you check whether the address has been given approval for this NFT in the `idToApprovals` variable or whether the address is an operator or not (the address has been given permission to transfer all NFT tokens) in the `ownerToOperators` variable. Also, you check whether the caller of this function is an owner of the NFT itself, which should be allowed to transfer the NFT.

Then, you add a function to add an NFT to the owner by adding the following code:

```
@internal
def _addTokenTo(_to: address, _tokenId: uint256):
    assert self.idToOwner[_tokenId] == empty(address)
    self.idToOwner[_tokenId] = _to
    self.ownerToNFTokenCount[_to] += 1
```

Inside the function, you add the owner of the NFT to the `idToOwner` variable. Then, you increase the owner's number of NFT in the `ownerToNFTokenCount` variable.

Then, you add a function to do the reverse thing by adding the following code:

```
@internal
def _removeTokenFrom(_from: address, _tokenId: uint256):
    assert self.idToOwner[_tokenId] == _from
    self.idToOwner[_tokenId] = empty(address)
    self.ownerToNFTokenCount[_from] -= 1
```

Inside the function, you set the empty address value to the owner of the NFT token ID in the `idToOwner` variable. Then, you decrease the number of NFT tokens of the owner in the `ownerToNFTokenCount` variable.

Then, you add the following function to clear the approvals:

```
@internal
def _clearApproval(_owner: address, _tokenId: uint256):
    assert self.idToOwner[_tokenId] == _owner
    if self.idToApprovals[_tokenId] != empty(address):
        self.idToApprovals[_tokenId] = empty(address)
```

Inside the function, you set the empty address value for the NFT token ID in the `idToApprovals` variable.

Then, you create a function to transfer an NFT by adding the following code:

```
@internal
def _transferFrom(_from: address, _to: address, _tokenId: uint256,
_sender: address):
    assert self._isApprovedOrOwner(_sender, _tokenId)
    assert _to != empty(address)
    self._clearApproval(_from, _tokenId)
    self._removeTokenFrom(_from, _tokenId)
    self._addTokenTo(_to, _tokenId)
    log Transfer(_from, _to, _tokenId)
```

Inside the function, you make sure the sender is able to transfer the NFT with the `_isApprovedOrOwner` function. Then, you clear the approvals with the `clearApproval` function. After that, you remove the NFT with the `_removeTokenFrom` function and add the NFT with the `_addTokenTo function`. Lastly, you log the `Transfer` event.

The previous function is an internal function. So, you need to create an external function to call the internal function by adding the following code:

```
@external
@payable
def transferFrom(_from: address, _to: address, _tokenId: uint256):
    self._transferFrom(_from, _to, _tokenId, msg.sender)
```

The function basically calls the `_transferFrom` function. This function is used if you want to transfer an NFT to a user's address, but not a smart contract.

Then, you create another function that can transfer an NFT to a smart contract by adding the following code:

```
@external
@payable
def safeTransferFrom(
        _from: address,
        _to: address,
        _tokenId: uint256,
        _data: Bytes[1024]=b""
    ):
    self._transferFrom(_from, _to, _tokenId, msg.sender)
    if _to.is_contract:
        returnValue: bytes4 = ERC721Receiver(_to).
onERC721Received(msg.sender, _from, _tokenId, _data)
        assert returnValue == method_
id("onERC721Received(address,address,uint256,bytes)", output_
type=bytes4)
```

Inside the function, you call the `_transferFrom` function as usual. Then, you check whether the destination address is a smart contract or not. If yes, you call the `onERC721Received` method on that smart contract and make sure the smart contract has the method.

Then, you create an external function to approve an NFT so a user can transfer it on behalf of the owner by adding the following code:

```
@external
@payable
def approve(_approved: address, _tokenId: uint256):
    owner: address = self.idToOwner[_tokenId]
    assert owner != empty(address)
    assert _approved != owner
    senderIsOwner: bool = self.idToOwner[_tokenId] == msg.sender
    senderIsApprovedForAll: bool = (self.ownerToOperators[owner])[msg.
sender]
    assert (senderIsOwner or senderIsApprovedForAll)
    self.idToApprovals[_tokenId] = _approved
    log Approval(owner, _approved, _tokenId)
```

Inside the function, you need to make sure the owner of the NFT has a valid address. Then, you need to ensure that the sender address is the operator or has been approved. After that, you change the value of the `idToApprovals` variable for the NFT token ID. Lastly, you log the `Approval` event.

Then, for approving an address to transfer all NFT on behalf of the owner, you add the following function:

```
@external
def setApprovalForAll(_operator: address, _approved: bool):
    assert _operator != msg.sender
    self.ownerToOperators[msg.sender][_operator] = _approved
    log ApprovalForAll(msg.sender, _operator, _approved)
```

Inside the function, you change the value of the `ownerToOperators` variable for the owner and the operator. If the value is `true`, then the operator can transfer all NFT tokens on behalf of the owner. To disallow it, you call this function with the `false` value. Lastly, you log the `ApprovalForAll` event.

Then, you need to create a function to mint an NFT by adding the following code:

```
@external
def mint(_to: address, _tokenId: uint256) -> bool:
    assert msg.sender == self.minter
    assert _to != empty(address)
    self._addTokenTo(_to, _tokenId)
    log Transfer(empty(address), _to, _tokenId)
    return True
```

To create an NFT, you make sure the sender is the minter. Then, you make sure the destination address that will become the owner of the NFT is not an empty address. Then you call the `_addTokenTo` function. After that, you log the `Transfer` event.

You also have to create a function to destroy an NFT by adding the following code:

```
@external
def burn(_tokenId: uint256):
    assert self._isApprovedOrOwner(msg.sender, _tokenId)
    owner: address = self.idToOwner[_tokenId]
    assert owner != empty(address)
    self._clearApproval(owner, _tokenId)
    self._removeTokenFrom(owner, _tokenId)
    log Transfer(owner, empty(address), _tokenId)
```

Inside the function, you make sure the sender has been approved or is the owner of the NFT. Then, you clear the approvals by calling the `_clearApproval` function. After that, you remove the NFT by calling the `_removeTokenFrom` function. Lastly, you log the `Transfer` event. In other words, destroying an NFT means transferring an NFT to an empty address.

The last function you need to implement is the `tokenURI` function:

```
@view
@external
def tokenURI(tokenId: uint256) -> String[132]:
    return concat(self.baseURL, uint2str(tokenId))
```

Remember the value of the `baseURL` variable? It's `https://packtpub.com/metadata/`. So, for the NFT token with an ID of `1`, for example, the tokenURI function will return `https://packtpub.com/metadata/1`. This is the URL of the NFT asset for the NFT token with an ID of `1`. Of course, you're responsible for putting the NFT asset in the URL.

Your NFT smart contract is complete. You can compile the smart contract by calling the following command:

```
(.venv) $ ape compile
```

To understand the flow of the NFT smart contract, let's create tests for the smart contract.

Unit tests

First, you need to update the fixture file. Edit `tests/conftest.py` and add the following code:

```
@pytest.fixture
def erc721_contract(deployer, project):
    return deployer.deploy(project.HelloNFT)

@pytest.fixture
def mint_nfts(erc721_contract, deployer, num_nfts=5):
    for i in range(num_nfts):
        erc721_contract.mint(deployer, i, sender=deployer)
    return num_nfts
```

The `erc721_contract` function is the fixture for your standard ERC-721 smart contract. To make the testing easier, you create a fixture function to mint five NFT tokens, named `mint_nfts`.

Then, you need to create a test file named `test_HelloNFT.py` inside the `tests` folder and then add the following code to it:

```
from ape.exceptions import ContractLogicError
import pytest

def test_init(erc721_contract, deployer):
    assert "Hello NFT" == erc721_contract.name()
    assert "HEL" == erc721_contract.symbol()

def test_balanceOf(erc721_contract, deployer, mint_nfts):
```

```
    assert 5 == erc721_contract.balanceOf(deployer)

def test_ownerOf(erc721_contract, deployer, mint_nfts):
    for i in range(5):
        assert deployer == erc721_contract.ownerOf(i)
```

The `test_init` function tests the initialization function of the smart contract and makes sure the `name` and `symbol` variables are initialized properly. The `test_balanceOf` tests the `balanceOf` function by checking the number of NFT tokens the deployer has. The `test_ownerOf` tests the `ownerOf` function by checking the owner of all NFT tokens inside the smart contract. Notice that the `test_balanceOf` and `test_ownerOf` functions use the `mint_nfts` fixture because they need five NFT tokens to be created first.

Now, let's add the `test_transfer` function by adding the following code:

```
def test_transfer(erc721_contract, deployer, mint_nfts, accounts):
    user = accounts[1]
    nftId = 2
    assert 5 == erc721_contract.balanceOf(deployer)
    assert 0 == erc721_contract.balanceOf(user)
    assert deployer == erc721_contract.ownerOf(nftId)
    erc721_contract.transferFrom(deployer, user, nftId,
sender=deployer)
    assert 4 == erc721_contract.balanceOf(deployer)
    assert 1 == erc721_contract.balanceOf(user)
    assert user == erc721_contract.ownerOf(nftId)
```

The `test_transfer` function tests the `transferFrom` function by checking the amount of NFT tokens the sender and the destination have before and after the execution of the function. You also test the function by checking the ownership of the NFT token that's being transferred before and after the execution of the function.

Now, let's add the `test_approve` function by adding the following code:

```
def test_approve(erc721_contract, deployer, mint_nfts, accounts):
    user1 = accounts[1]
    user2 = accounts[2]
    nftId = 2
    otherNftId = 3

    assert erc721_contract.getApproved(nftId) == '0x00000000000000000000
0000000000000000000'
    assert erc721_contract.getApproved(otherNftId) == '0x0000000000000
00000000000000000000000000'
    with pytest.raises(ContractLogicError):
```

```
        erc721_contract.transferFrom(deployer, user2, nftId,
sender=user1)
        erc721_contract.transferFrom(deployer, user2, otherNftId,
sender=user1)

    erc721_contract.approve(user1, nftId, sender=deployer)
    assert erc721_contract.getApproved(nftId) == user1
    assert erc721_contract.getApproved(otherNftId) == '0x0000000000000
0000000000000000000000000'

    assert deployer == erc721_contract.ownerOf(nftId)
    erc721_contract.transferFrom(deployer, user2, nftId, sender=user1)
    assert user2 == erc721_contract.ownerOf(nftId)

    with pytest.raises(ContractLogicError):
        erc721_contract.transferFrom(deployer, user2, otherNftId,
sender=user1)
```

In the `test_approve` function, you test the `approve` function by checking whether an address that has been approved can transfer an NFT on behalf of the owner. You test the `getApproved` function as well. You make sure to test that the `approve` function only gives the approval for one NFT.

Now, let's add the `test_setApprovalForAll` function by adding the following code:

```
def test_setApprovalForAll(erc721_contract, deployer, mint_nfts,
accounts):
    user1 = accounts[1]
    user2 = accounts[2]
    nftId = 2
    otherNftId = 3

    with pytest.raises(ContractLogicError):
        erc721_contract.transferFrom(deployer, user2, nftId,
sender=user1)
        erc721_contract.transferFrom(deployer, user2, otherNftId,
sender=user1)

    erc721_contract.setApprovalForAll(user1, True, sender=deployer)

    assert deployer == erc721_contract.ownerOf(nftId)
    erc721_contract.transferFrom(deployer, user2, nftId, sender=user1)
    assert user2 == erc721_contract.ownerOf(nftId)

    assert deployer == erc721_contract.ownerOf(otherNftId)
    erc721_contract.transferFrom(deployer, user2, otherNftId,
sender=user1)
    assert user2 == erc721_contract.ownerOf(otherNftId)
```

In this test function, you test the `setApprovalForAll` function by checking that after the function's execution, you can transfer two NFT tokens, not just one.

To execute the test file, run the following command:

```
(.venv) $ py.test tests/test_HelloNFT.py
============================================= test session starts
=============================================
platform darwin -- Python 3.10.12, pytest-7.4.0, pluggy-1.3.0
rootdir: /Users/arjuna/Code/book/Hands-On-Blockchain-for-Python-
Developers--2nd-Edition/chapter_13/nft
plugins: web3-6.9.0, eth-ape-0.6.18
collected 6 items

tests/test_HelloNFT.py ......
                                                [100%]

============================================== 6 passed in 2.87s
==============================================
```

Your NFT smart contract has been successfully tested. You can go and deploy it on the blockchain.

Summary

In this chapter, you have learned how to create NFT on top of Ethereum. First, you created a very simple NFT smart contract and you learned why NFT technology offers something valuable compared to traditional applications. Then, you created the ERC-721 standard smart contract by implementing methods required by the ERC-721 standard. You learned what the ERC-721 standard is and why it's needed. In the next chapter, you'll learn how to create an NFT exchange smart contract where you can trade your NFT tokens.

Part 6: Writing Complex Smart Contracts

This section delves into the realm of intricate smart contract development, equipping you with the knowledge and skills to navigate complex blockchain ecosystems. First, you will learn the architecture and mechanics behind creating a decentralized marketplace for non-fungible tokens. This will empower you to facilitate digital asset exchange with transparency. Then you will embark on a journey to construct a lending protocol, enabling borrowing and lending with a vault. Finally, you will explore the intricacies of building an automated market maker (AMM) decentralized exchange (DEX), which will enable you to facilitate seamless and trustless asset swaps on the blockchain.

This section has the following chapters:

- *Chapter 14*, *Writing NFT Marketplace Smart Contracts*
- *Chapter 15*, *Writing a Lending Vault Smart Contract*
- *Chapter 16, Decentralized Exchange*

14
Writing NFT Marketplace Smart Contracts

In this chapter, you're going to learn about the NFT marketplace. You've written an ERC-721 NFT smart contract in the previous chapter. You also even added some functions to sell NFTs for ETH. Now, you want to go further and create an NFT marketplace where people can trade their NFTs. This marketplace smart contract will interact with the NFT smart contract you've written previously.

The following topics will be covered in this chapter:

- Understanding the NFT marketplace
- Writing the NFT marketplace smart contract
- Writing the tests
- Enhancing the NFT marketplace

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_14`.

Understanding the NFT marketplace

The **NFT marketplace** is just like any marketplace. One of the examples of a marketplace is Amazon, where people can sell and buy things. As a buyer, you can search for any items you like, from books and smartphones to kitchen appliances, and so on. But you can also be a seller. You can upload information about your items to sell on Amazon. Then people can buy the items from you, and you receive the payment from the buyer.

To be exact, the process could be a bit different. Instead of paying you directly, the buyer sends the payment to the marketplace first. After you deliver the goods, the marketplace releases the payment to you from the buyer. You can say the marketplace acts as an escrow so the seller and the buyer can do trading safely. Without the marketplace, the seller and the buyer can still do transactions, but they might not trust each other. Imagine, the buyer could send money to the seller, but then the seller refuses to send the goods. The buyer is being cheated.

The marketplace's role is to build trust in trading. Sellers will be dissuaded from cheating the buyers in the knowledge that the marketplace could punish them. For example, if the seller doesn't deliver the goods, the buyer can complain to the marketplace and the marketplace will refund the money to the buyer. On top of that, the marketplace could ban the seller to protect other buyers. This way, the marketplace builds trust between buyers and sellers.

Marketplaces also give other benefits to sellers and buyers alike. If the marketplace is very popular, it will have a lot of sellers and buyers. For buyers, this is good news because the selection of items to choose from will be a lot wider, from laptops to books and more. But a higher number of sellers will give you an even better chance of finding niche items such as a magic wand or Ouija board, for example.

From the seller's point of view, the bigger the marketplace, the better chance they have of selling their items. Many buyers will flock to big marketplaces. If you sell particularly niche items, then there will probably be at least a few among this big number of buyers who want to buy them.

Without a marketplace to go to, as a buyer, you have to spend an incredible amount of time to find sellers who sell magic wands made from gold. If you have a lot of things to buy, you'll need to find numerous sellers located across various areas. It's more efficient if they're in one place such as a marketplace. In a way, the purpose of a marketplace is to become a place for sellers and buyers to gather.

In this gathering place, buyers and sellers can bargain. The seller can propose a price for an item. If the price is considered expensive, the buyer can bargain and propose a lower price. It's up to them whether they want to accept the price.

As a regulator, the marketplace can enact regulations to protect the quality of items. This is to protect those buyers who don't have the capabilities or time to assess the quality of the items offered by the sellers. So, buyers coming to the marketplace have a guarantee that the items in the marketplace pass a certain threshold standard of quality.

To protect buyers further, the marketplace can ban certain dangerous products such as guns, poisons, protected animals, and so on. To help buyers with shopping, the marketplace can specialize in one category of things, such as wet markets where people can buy and sell fish and other items from the sea.

To improve the experience of the buyers, the marketplace can offer a *search* feature where buyers can find items easily. This is especially true in online spaces where buyers can type the items they want to buy and get the search results immediately.

To make the experience even better, the marketplace can offer a *recommended items* feature. If a buyer buys certain items, such as a laptop, they will get recommended similar items, such as a mouse, a cleaning tool to wipe the screen of the laptop, or laptop bags. The marketplace can use AI technology to compile the recommended items based on the buyers' habits.

However, these things come with a cost. The marketplace has the power to do things that are harmful to buyers and sellers. Suppose the owner of the marketplace has a left-wing political leaning and a buyer wants to sell right-wing political books, the owner could ban the books from being sold. If that's not possible, the owner could undermine the visibility of the books. That way, the book sales could plummet.

Now that you have the basic idea of what a marketplace entails, let's come back to the concept of the NFT marketplace. An NFT marketplace is a place where buyers and sellers trade their NFTs. This definition is like the definition of the marketplace you've just learned, be it online or traditional.

So, what's the big deal with NFT marketplaces? Why can't we trade NFTs in an online marketplace such as Amazon? You can. Technically, you can trade NFTs in an online marketplace. But the downside is that you need to put trust in the marketplace. In blockchain, you want to be as trustless as possible. You want to be able to put trust in the smart contract or the program, not in the human operator of the marketplace.

An NFT marketplace in the form of a smart contract offers something valuable, such as ERC-20 tokens or ERC-721 NFTs. When you trade NFTs in a smart contract, you can inspect the rules of the marketplace in the smart contract and decide whether they are fair or not. So, smart contracts bring transparency to the marketplace. In a traditional online marketplace, you don't know why certain decisions are made because the system is not transparent.

You can create a smart contract where not even the deployer or the owner of the smart contract can censor transactions between sellers and buyers. Of course, you can also create a smart contract where you can ban or freeze accounts that you don't like. But the difference is that users can inspect your smart contract and see that rule. Then it's up to them whether they still want to interact with your smart contract or not.

NFT marketplaces are smart contracts. The NFTs that people trade in the NFT marketplace are also smart contracts. So, in this case, we will have interactions between smart contracts. In previous chapters, the user of a smart contract has always been a person or an externally owned account address, but the user of a smart contract can be another smart contract.

First, how do you build an NFT marketplace smart contract? Not the way you build a traditional online marketplace. In the traditional way, the transaction flow will look like this. A seller of an NFT must transfer the NFT to the marketplace so the ownership changes to the marketplace. A buyer who wants to purchase the NFT must transfer the payment in ETH to the marketplace. Once the marketplace receives the payment in ETH, the marketplace will transfer the NFT to the buyer and transfer ETH minus the fee to the seller.

You could build the marketplace this way and there are NFT marketplaces that operate this way. They're called centralized NFT marketplaces. One of the examples is the Binance NFT marketplace. It can work, and that's fine. But you want to build your NFT marketplace in a decentralized way.

There is a way in a smart contract where the marketplace can receive the payment from the buyer and forward it to the seller after transferring the NFT from the seller to the buyer in one go.

The magic is in the `transferFrom` function from the ERC-721 standard along with the `approve` function. If you want to sell your NFT in an NFT marketplace, you as the owner of NFT should give approval to the NFT marketplace for the NFT you want to sell with the `approve` function from the NFT smart contract. Then, inside the function where a user buys an NFT from you in the marketplace, the `transferFrom` function is used to transfer the NFT from you as the seller to the buyer. It's the NFT marketplace that invokes the `transferFrom` function. Once the NFT marketplace accepts the payment from the buyer, it can transfer the payment directly to the seller.

In short, the sale of an NFT in an NFT marketplace is composed of these steps:

- The payment is received from the buyer
- The marketplace transfers the NFT from the seller to the buyer using the `transferFrom` function
- The payment is forwarded to the seller

The requirement is that the seller must give approval to the NFT marketplace before selling the NFT. This is the secret sauce of an NFT marketplace. A seller doesn't need to transfer NFTs to the marketplace to sell their NFTs. The seller only needs to give approval to the NFT marketplace.

But what if the NFT marketplace steals your NFT?

If you give approval to someone using the `approve` function, it means the person can use the `transferFrom` function to transfer your NFT to them. They can steal your NFT – but only if the approval is given to an externally owned account or an address that's owned by a person. But the NFT marketplace is a smart contract, not a person. It doesn't have its own will. It's just a program.

Can a program or a smart contract steal your NFT, even theoretically? Yes, it can. A smart contract can have a backdoor function built in where the owner of the NFT smart contract can use the approval you've given to steal your NFT.

But you can audit NFT marketplace smart contracts to make sure no such backdoor functions exist. A smart contract is neutral. It's just code. A person can write a smart contract to steal NFTs, and they can also write a smart contract where nobody can steal NFTs. Other than those two possibilities, smart contracts could have an unintentional bug where someone can drain your NFTs through the approval you've given. If that happens, you can revoke the approval so the NFT marketplace doesn't have permission to transfer your NFT anymore.

To understand this fully, let's write an NFT marketplace smart contract ourselves.

Writing the NFT marketplace smart contract

As usual, let's create the *Ape project* by running these commands:

```
$ python3 -m venv .venv
(.venv) $ pip install eth-ape'[recommended-plugins]'==0.7.23
(.venv) $ mkdir nft_marketplace
(.venv) $ cd nft_marketplace
(.venv) $ ape init
Please enter project name: NFT Marketplace
SUCCESS: NFT Marketplace is written in ape-config.yaml
```

Then create a smart contract file inside the `contracts` folder named `NFTMarketplace.vy` and add the following code to it:

```
#pragma version ^0.3.0

interface ERC721_Interface:
    def transferFrom(_from: address, _to: address, _tokenId: uint256):
nonpayable
    def ownerOf(_tokenId: uint256) -> address: view
```

The `pragma` line specifies that this smart contract can only be compiled using version 0.3.x of the Vyper compiler.

Then there are two ERC721 interfaces that you register in the smart contract because the NFT marketplace smart contract will interact with the NFT smart contract. However, you only use two functions from the NFT marketplace: the `transferFrom` and `ownerOf` functions. So, no other ERC-721 functions are registered in the interface.

Then add the state variables of the NFT marketplace smart contract by adding the following code:

```
prices: public(HashMap[address, HashMap[uint256, uint256]])

proposals: public(HashMap[address, HashMap[uint256, HashMap[address,
uint256]]])
```

The `prices` variable refers to the prices set by the sellers of NFTs. The address key points to the NFT smart contract address. The inner `HashMap` maps the token ID in the NFT smart contract to the price.

The `proposals` variable refers to the proposed prices set by the NFT buyers. The key address also points to the NFT smart contract address. The inner key refers to the token ID in the NFT smart contract. Then the innermost key refers to the buyer's address. The value of the most inner `HashMap` is the price.

Next, add the function to buy the NFT that the owner wants to sell by adding the following code:

```
@external
@nonreentrant("lock")
@payable
def buyNFT(nftAddress: address, tokenId: uint256):
    assert self.prices[nftAddress][tokenId] != 0

    assert msg.value >= self.prices[nftAddress][tokenId]

    buyer: address = msg.sender

    nftContract: ERC721_Interface = ERC721_Interface(nftAddress)

    seller: address = nftContract.ownerOf(tokenId)

    nftContract.transferFrom(seller, buyer, tokenId)

    send(seller, self.prices[nftAddress][tokenId])

    if msg.value > self.prices[nftAddress][tokenId]:
        send(buyer, msg.value - self.prices[nftAddress][tokenId])

    self.prices[nftAddress][tokenId] = 0
```

The function accepts two arguments, the NFT smart contract address and the NFT token ID. In this function, you make sure the price has been set by the owner of the NFT by checking the value in the `prices` variable. Then you make sure the ETH sent to this smart contract through this function is at least the same as the price of the NFT or bigger. After that, you transfer the NFT in the NFT smart contract from the seller to the buyer with the `transferFrom` function. The next step is to transfer the payment from the buyer to the seller for the price of the NFT. The final step is to reset the price of the NFT to zero. You assume that the new owner of the NFT doesn't want to sell their new NFT.

Notice that the NFT marketplace smart contract interacts with the NFT smart contract using the `ERC721_Interface` interface. It creates the smart contract using the interface with the NFT smart contract address by using the following code:

```
nftContract: ERC721_Interface = ERC721_Interface(nftAddress)
```

Then you can call the `transferFrom` function from this object.

Notice also that this function is protected by the `nonreentrant` annotation. This is a good idea because this function sends ETH using the `send` method. The buyer of the NFT can be a smart contract and a smart contract can receive ETH using a function that has some nasty logic, such as calling the function to buy the NFT recursively.

But how does the price show up in the `prices` variable? You need to add the function to set the price for the owner by adding the following code:

```
@external
def setNFTPrice(nftAddress: address, tokenId: uint256, price:
uint256):
    nftContract: ERC721_Interface = ERC721_Interface(nftAddress)

    assert nftContract.ownerOf(tokenId) == msg.sender

    self.prices[nftAddress][tokenId] = price
```

The function accepts three arguments, which are the NFT smart contract address, the NFT token ID, and the price. Inside the function, you can check the ownership of the NFT using the `ownerOf` function from the NFT smart contract you create using the `ERC721_Interface` interface. Then you set the price of the NFT you want to sell in the `prices` variable.

Let us consider a situation in which the owner of the NFT is too passive and doesn't have the urge to sell the NFT. However, let's assume that the buyer still wants to buy the NFT. In such a situation, you want to create the possibility for the buyer to offer a price to the owner of the NFT to see whether they're willing to sell it or not. Whether the owner of the NFT accepts the offer or not is up to them. Create a function so the buyer can offer to buy the NFT from the owner by adding the following code:

```
@external
@payable
def proposeNFTPrice(nftAddress: address, tokenId: uint256,
proposedPrice: uint256):
    assert msg.value == proposedPrice, "ETH is not same as proposed
price"

    self.proposals[nftAddress][tokenId][msg.sender] = proposedPrice
```

The preceding function accepts three arguments, which are the NFT smart contract address, the NFT token ID, and the proposed price for the NFT. It makes sure the ETH a buyer sends to the smart contract through this function matches the proposed price. Then you set the proposed price in the `proposals` variable that records the buyer's address as well in the innermost key.

But after a buyer proposes a price for the NFT, sometimes they might change their mind. So you need to add a function to cancel the offer by adding the following code:

```
@external
@nonreentrant("lock2")
def cancelProposalNFTPrice(nftAddress: address, tokenId: uint256):
    proposedPrice: uint256 = self.proposals[nftAddress][tokenId][msg.
sender]
```

```
    assert proposedPrice > 0, "Proposed price is zero"

    self.proposals[nftAddress][tokenId][msg.sender] = 0

    send(msg.sender, proposedPrice)
```

This function accepts two arguments, which are the NFT smart contract address and the NFT token ID. First, you make sure the proposed price has been set by checking the value is bigger than zero. Then you set the proposed price of the buyer in the `proposals` variable to zero. Finally, you send back the ETH that the buyer has sent to the smart contract.

The last function you need to add is one that allows the owner of the NFT to review and accept an offer made by a potential buyer, if they deem the offered price to be satisfactory. To help the owner accept the offer, you should create the function to accept the NFT offer by adding the following code:

```
@external
@nonreentrant("lock3")
def acceptNFTProposal(nftAddress: address, tokenId: uint256, buyer:
address):
    nftContract: ERC721_Interface = ERC721_Interface(nftAddress)

    assert nftContract.ownerOf(tokenId) == msg.sender

    assert self.proposals[nftAddress][tokenId][buyer] != 0

    proposedPrice: uint256 = self.proposals[nftAddress][tokenId]
[buyer]

    nftContract.transferFrom(msg.sender, buyer, tokenId)

    send(msg.sender, proposedPrice)

    self.proposals[nftAddress][tokenId][buyer] = 0

    self.prices[nftAddress][tokenId] = 0
```

This function accepts three arguments, which are the NFT smart contract address, the NFT token ID, and the buyer's address. Inside the function, you check the ownership of the owner of the NFT using the `ERC721_Interface` interface and make sure it's the same as the caller of the function. Then you make sure the proposed price from the buyer exceeds zero. Then you transfer the NFT from the owner to the buyer using the `transferFrom` function of the NFT smart contract. Finally, you send ETH from the smart contract to the owner. Remember, previously the buyer had to send ETH to the smart contract when proposing a price or creating an offer for the NFT.

You can compile the smart contract by executing the following command:

```
(.venv) $ ape compile
```

Now that you have the smart contract ready, let's simulate a trade of NFTs with unit tests.

Writing the tests

Before you jump into testing the NFT marketplace, you need an NFT to trade! Testing your NFT marketplace smart contract requires you to have an NFT smart contract as well. You can get the NFT smart contract from the previous chapter or download it from `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_14/nft_marketplace/contracts/HelloNFT.vy`.

Put `HelloNFT.vy` inside the `contracts` folder, then compile the NFT smart contract as follows:

```
(.venv) $ ape compile
```

Then let's set up the fixture test file by creating a file named `conftest.py` inside the tests folder and add the following code to it:

```
import pytest


@pytest.fixture
def deployer(accounts):
    return accounts[0]

@pytest.fixture
def marketplace_contract(deployer, project):
    return deployer.deploy(project.NFTMarketplace)

@pytest.fixture
def nft_contract(deployer, project):
    return deployer.deploy(project.HelloNFT)

@pytest.fixture
def mint_nfts(nft_contract, deployer, num_nfts=5):
    for i in range(num_nfts):
        nft_contract.mint(deployer, i, sender=deployer)
    return num_nfts
```

The fixture file does a couple of things:

1. The `deployer` fixture function returns the deployer of the smart contract, which is the first account.
2. The `marketplace_contract` function and the `nft_contract` function use the `deployer` function to deploy the smart contract.
3. The `marketplace_contract` function returns the NFT marketplace smart contract object and the `nft_contract` function returns the NFT smart contract, or in this case, the `HelloNFT` smart contract.
4. The `mint_nfts` function is a fixture function to mint 5 NFTs in the NFT smart contract.

Create a test file named `test_NFTMarketplace.py` inside the `tests` folder and add the following code:

```
from ape.exceptions import ContractLogicError
import pytest


def test_setNFTPrice(marketplace_contract, nft_contract, mint_nfts,
deployer, accounts):
    tokenId = 1
    price = 1000
    marketplace_contract.setNFTPrice(nft_contract.address, tokenId,
price, sender=deployer)
    assert marketplace_contract.prices(nft_contract.address, tokenId)
== price
```

This test function tests the `setNFTPrice` function by checking the prices variable. This test function uses two smart contracts, which are the NFT smart contract and the NFT marketplace smart contract. The first argument of the `setNFTPrice` function is the address of the NFT smart contract address.

Then you add the test function of `buyNFT` by adding the following code:

```
def test_buyNFT(marketplace_contract, nft_contract, mint_nfts,
deployer, accounts):
    tokenId = 1
    price = 1000
    user = accounts[1]
    initial_balance_deployer = deployer.balance
    initial_balance_user = user.balance
    marketplace_contract.setNFTPrice(nft_contract.address, tokenId,
price, sender=deployer)
    nft_contract.approve(marketplace_contract.address, tokenId,
sender=deployer)
    marketplace_contract.buyNFT(nft_contract.address, tokenId,
value=price, sender=user)
```

```
    assert marketplace_contract.prices(nft_contract.address, tokenId)
== 0
    assert nft_contract.ownerOf(tokenId) == user
```

Inside the `test_buyNFT` test function, there are a couple of things happening:

1. The owner of the NFT sets the price of the NFT with the `setNFTPrice` function.
2. Then the owner gives approval to the NFT marketplace with the `approve` function of the NFT smart contract. The result is that the NFT marketplace can transfer the given NFT from the owner.
3. In the next step, the buyer buys the NFT with the `buyNFT` function. The buyer must send ETH when executing this function.
4. Then you check the price of the NFT and make sure it has been set to zero in the `prices` variable.
5. Lastly, you check the ownership of the NFT from the NFT smart contract by using the `ownerOf` function. The new owner should be the buyer.

Then let's add the test for the NFT offer:

```
def test_proposeNFTPrice(marketplace_contract, nft_contract, mint_
nfts, deployer, accounts):
    tokenId = 1
    price = 2000
    user = accounts[1]
    marketplace_contract.proposeNFTPrice(nft_contract.address,
tokenId, price, value=price, sender=user)
    assert marketplace_contract.proposals(nft_contract.address,
tokenId, user) == price
```

In this function, the user calls the `proposeNFTPrice` function with the price the user wants for the NFT that comes from an NFT smart contract. The user also needs to send ETH, as you can see in the `value` argument. To check whether the `proposeNFTPrice` function works or not, you check the `proposals` variable and make sure the value is correct for the NFT smart contract, the NFT, and the user.

To test the *cancel offer* function, add the following code:

```
def test_cancelProposalNFTPrice(marketplace_contract, nft_contract,
mint_nfts, deployer, accounts):
    tokenId = 1
    price = 2000
    user = accounts[1]
    marketplace_contract.proposeNFTPrice(nft_contract.address,
tokenId, price, value=price, sender=user)
    marketplace_contract.cancelProposalNFTPrice(nft_contract.address,
```

```
tokenId, sender=user)
    assert marketplace_contract.proposals(nft_contract.address,
tokenId, user) == 0
```

In the `test_cancelProposalNFTPrice` function, the user calls the `proposeNFTPrice` function and then calls the `cancelProposalNFTPrice` function. To make sure the `cancelProposalNFTPrice` function works, you need to check the `proposals` variable and make sure the value is zero for the NFT smart contract, the NFT, and the user.

The last function you need to add is a function to test the *accept NFT offer* function:

```
def test_acceptNFTProposal(marketplace_contract, nft_contract, mint_
nfts, deployer, accounts):
    tokenId = 1
    price = 2000
    user = accounts[1]
    nft_contract.approve(marketplace_contract.address, tokenId,
sender=deployer)
    marketplace_contract.proposeNFTPrice(nft_contract.address,
tokenId, price, value=price, sender=user)
    marketplace_contract.acceptNFTProposal(nft_contract.address,
tokenId, user, sender=deployer)
    assert nft_contract.ownerOf(tokenId) == user
```

Inside the `test_acceptNFTProposal` function, the deployer or the owner of the NFT approves the NFT marketplace to be able to transfer the NFT with the function of the NFT smart contract. Then the user offers to buy the NFT with the `proposeNFTPrice` function of the NFT marketplace. After that, the owner accepts the offer with the `acceptNFTProposal` function. Lastly, you check that the ownership of the NFT has changed using the `ownerOf` function. The new owner should be the user who offered to buy the NFT.

To test the NFT marketplace smart contract, run the following command:

```
(.venv) $ py.test tests/test_NFTMarketplace.py
============================================= test session starts
=============================================
platform darwin -- Python 3.10.12, pytest-7.4.0, pluggy-1.3.0
rootdir: /Users/arjuna/Code/book/Hands-On-Blockchain-for-Python-
Developers--2nd-Edition/chapter_14/nft_marketplace
plugins: web3-6.9.0, eth-ape-0.6.18
collected 5 items

tests/test_NFTMarketplace.py .....
                                                    [100%]

============================================== 5 passed in 2.88s
==============================================
```

Your NFT marketplace smart contract is good to go, but there are some improvements you can consider before you launch the NFT marketplace.

Enhancing the NFT marketplace

Here are some ideas for you to implement in your NFT marketplace application. They can make your NFT marketplace more convenient, generate revenue, and make users happier.

Supporting non-standard NFTs

The NFT marketplace smart contract you've written interacts with the `HelloNFT` smart contract. But it can support other NFT smart contracts. People can buy and sell other NFTs from other NFT smart contracts. One requirement exists, however: any NFT smart contract that wants to be traded in this NFT marketplace smart contract needs to follow the ERC-721 standard. As you can see, you use the `transferFrom` function of the NFT smart contract to transfer the NFT from the seller to the buyer. You also use the `ownerOf` function of the NFT smart contract to check the ownership of the NFT.

But what if the NFT smart contract doesn't follow the ERC-721 standard?

You can just say, too bad. It's their fault for not following the standard. But what if that particular NFT smart contract is very popular? You can omit it from your marketplace, but people will just trade it in another NFT marketplace. Your loss.

To solve this, you can always make an exception. You can create an adapter for NFT smart contracts that don't follow the ERC-721 standard. The code looks like this in the simplest case:

```
If nftSmartContract.address == "0x03ff...":
    do_something_different(seller, buyer, tokenId):
else: # all other NFT smart contracts that follow standard
    transferFrom(seller, buyer, tokenId)
```

It's as easy as that. Although all new NFT smart contracts you'll encounter will probably follow the ERC-721 standard, there are some that don't. An example is the **CryptoPunks NFT** collection, which is very popular and highly valued by many people. The smart contract doesn't follow the ERC-721 standard as you can verify in the source code at `https://github.com/larvalabs/cryptopunks/blob/master/contracts/CryptoPunksMarket.sol`. The reason for this is because it predates the ERC-721 standard. The CryptoPunks collection was born before the ERC-721 standard was written.

Supporting ERC-20 tokens

The NFT marketplace smart contract you've written only accepts ETH as the payment currency, but users might want to use other cryptocurrencies to buy and sell NFTs. There are many cryptocurrencies that are popular, such as USDC, USDT, DAI, and so on.

To support ERC-20 tokens in your NFT marketplace smart contract, the procedure is similar to the NFT trades. Right now, the owner of the NFT sells their NFT for ETH. The buyer buys NFT with ETH. But you can consider this process in reverse: the owner of NFT buys ETH with their NFT and the NFT buyer sells their ETH and accepts NFT as the payment.

Now, replace ETH with the ERC-20 tokens. The NFT owner buys ERC-20 tokens with their NFT or sells their NFT for ERC-20 tokens. So, you need to create a function that can swap an NFT for ERC-20 tokens. For the NFT marketplace smart contract to be able to transfer an NFT from the seller to the buyer, the seller needs to give approval to the NFT marketplace smart contract in the NFT smart contract.

So, what does the NFT marketplace need to be able to transfer ERC-20 tokens from the buyer to the seller?

The buyer needs to give approval to the NFT marketplace smart contract in the ERC-20 tokens smart contract so the NFT marketplace can spend the tokens on the buyer's behalf. Suppose the seller wants to sell their NFT for 100 USDC. Then the buyer must approve the NFT marketplace smart contract in the USDC smart contract so the NFT marketplace smart contract can transfer 100 USDC to the seller on the buyer's behalf!

Remember that the ERC-20 tokens smart contract follows the ERC-20 standard. The ERC-20 tokens smart contract has an `approve` function and a `transferFrom` function. The buyer of the NFT uses the `approve` function to give approval to the NFT marketplace smart contract. The NFT marketplace smart contract uses the `transferFrom` function to transfer ERC-20 tokens (or USDC in this example) from the buyer to the owner of NFT.

So, for illustration, in the NFT marketplace smart contract's trade function, you could have these lines of code:

```
def tradeNFT():
    nftSmartContract.transferFrom(seller, buyer, nftTokenId)
    usdcSmartContract.transferFrom(buyer, seller, 10000000)
```

Using ERC-20 tokens is much more convenient than using ETH when trading NFTs in the NFT marketplace. Remember that when a buyer wants to offer to buy an NFT from the owner of an NFT, the buyer needs to send their ETH to the NFT marketplace smart contract. The owner of the NFT will probably need time to decide whether to accept or not. While waiting for this, the ETH from the buyer is locked in the NFT marketplace smart contract. The buyer might want to use the ETH for something productive in the meantime. Using ERC-20 tokens, you don't have to lock your ERC-20 tokens in the NFT marketplace smart contract. You only need to give approval to the NFT marketplace smart contract.

Having said that, people still want to buy NFTs with ETH and want to have the convenience of using ERC-20 tokens when trading. To solve this problem, you can lock ETH in a WETH smart contract. **WETH** is short for **Wrapped ETH**, and is an ERC-20 token smart contract. 1 WETH is equal to 1 ETH. To get the WETH from the WETH smart contract, you need to send ETH to the WETH smart contract, and your ETH will be locked in the WETH smart contract. So, you can use WETH in the NFT marketplace smart contract just like you use ERC-721 tokens.

Later, if you or the NFT marketplace smart contract wants to get the ETH from the WETH, you can unwrap the ETH again from the WETH smart contract.

There are other improvements you can add to the NFT marketplace. For example, to generate revenue from your hard labor in writing the NFT marketplace smart contract, you could implement fees so every time there is a transaction, you get a cut from it. You need to make sure there is a function to withdraw the revenue from the smart contract for this.

Summary

In this chapter, you learned the concept of the marketplace and what makes NFT marketplaces special. You understood that the value proposition of NFT marketplaces is that NFT trades are transparent and users can put their trust not in the human operators, but in the smart contract or algorithms of the marketplace.

Then, you wrote the NFT marketplace smart contract and learned how to interact with another smart contract, which was the NFT smart contract in this case. After writing the smart contract, you wrote a unit test to learn how the NFT marketplace conducts the transaction between the NFT seller and the buyer.

Lastly, you thought about some improvements you could add to the NFT marketplace smart contract. Knowing how to build an NFT marketplace that interacts with other smart contracts is beneficial because now you can build a place where people can exchange their assets safely. On top of that, a lot of smart contracts in real life deal with other smart contracts, so this knowledge will be very beneficial even if you build other kinds of smart contracts not related to NFTs.

In the next chapter, you'll learn how to write a lending protocol smart contract with a vault.

15
Writing a Lending Vault Smart Contract

In this chapter, you're going to learn about two concepts, which are **lending** and **vaults**. First, you're going to learn about vaults. A vault is a place where people store their money and can take it back if they want to. The vault smart contract has a standard you can follow. After learning about vaults, you will learn about lending, where users can borrow assets from the lending smart contract and repay it with interest. Then, you will combine these two smart contracts into one lending vault smart contract. After learning how to write a lending vault smart contract, you will understand two of the most important and popular protocols in blockchain, which are lending and vaults.

The following topics will be covered in this chapter:

- Understanding vaults
- The ERC-4626 standard
- Lending

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_15`.

Understanding vaults

There are many cases where people treat smart contracts as vaults. What I mean is people store money in smart contracts. You've seen it in *Chapter 3*, *Using Vyper to Implement a Smart Contract*, where the owner of the smart contract solicits donations in the smart contract. However, a vault is not just any smart contract that can accept money. There must be a way for a user to withdraw their money from the vault. The money can be ETH or any other ERC-20 cryptocurrencies.

Note

A vault is not just a passive place where money is kept. A vault can be a way to pool resources. The resources can later be put into productive ventures to get a yield or profit.

To make things clearer, consider this case. User A deposits 1000 USDT into a vault smart contract. User B deposits 2000 USDT into the vault. User C deposits 500 USDT into the vault. Now, the vault has accumulated assets of as much as 3500 USDT. These assets can be put into many investments that give a 10% yield per year on average. The yield later can be divided fairly into User A, User B, and User C. This is more efficient because the users may not have time to find profitable investments. Also, some investments require certain thresholds of money. It could be that investment X needs at least 1000 USDT at minimum. User C and User A would be locked out from investment X. However, by pooling their resources, they can get their money into investment X. So, it's similar to a hedge fund or a bank.

To visualize the vault, you can take a look at the following image, where **User A** deposits **2 ETH**, **User B** deposits 1 ETH, and then **User A** withdraws **1 ETH**.

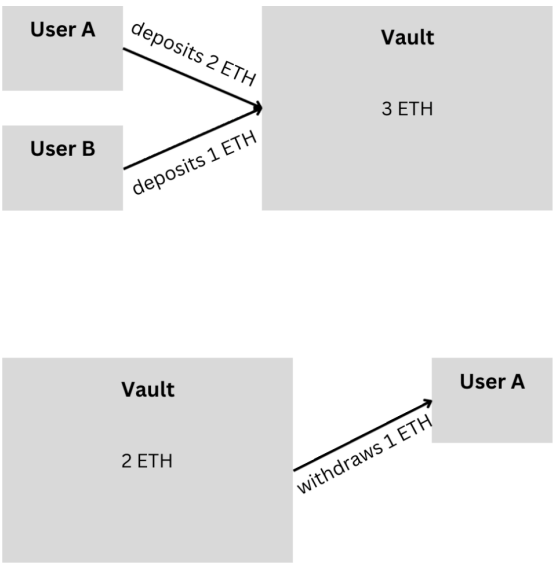


Figure 15.1: A vault

Of course, all of these can be achieved with a traditional bank or hedge fund. You send your assets, either cryptocurrency or USDT fiat money, to an institution. They will manage your assets along with other people's assets. Periodically, they will send a report to you on how your assets perform. Does it go up, go down, or move sideways? But you have to trust them. You don't know exactly whether they tell the truth or not. In this case, only the rule of law or legislation can give you peace of mind.

With the vault in the form of a smart contract, the management of assets becomes transparent. You can see how many assets the vault smart contract has. You can peek into their ventures or where your assets, along with other people's assets, are invested. Of course, you can audit the smart contract to make sure the owner of the smart contract cannot steal your assets blatantly.

Also, there is a programmable aspect of the smart contract. Since the vault is a smart contract or a program in the basic sense, you can put automation into it. In the traditional bank, there is a cashier that handles your deposit. However, a smart contract can accept deposits from anyone or even another program. You can set up a notification system where you get an alert if something happens in the vault smart contract. Traditional institutions such as banks or hedge funds are more discreet.

You can divide smart contract vaults into a few categories based on how they operate. They are **lending vaults**, **liquidity vaults**, and **staking vaults**. A lending vault pools assets for borrowers to borrow. A liquidity vault gathers assets to be deployed in investments. A staking vault is for staking assets such as staking ETH. But, of course, you can do whatever you wish with a vault smart contract.

A vault smart contract mechanism indicates deposit and withdrawal operations from users. If a user deposits 10 USD to the vault, the user should be able to withdraw or redeem their assets. If the assets have not been used at all, the user should be able to withdraw their assets fully minus the management fee. If 50% of the managed assets have been used for lending or locked in the investment, the user should be able to withdraw 50% of their assets minus the management fee, unless there is a locked period in the vault. If the vault smart contract makes profits, the user should be able to get some profits based on their contributions to the vault.

This brings us to the concept of **shares**. The more assets you deposit into the vault, the more shares you should get. These shares determine your pie of revenue being shared. The bigger shares you have, the bigger pie you should get. In blockchain, these shares can be conceptualized by **tokenization**. When you deposit assets to the vault smart contract, you should get the tokens from the vault smart contract. These tokens are the representation of your shares.

You can do many things with these tokens. You can trade them. You can use it as currency. For example, let's say you have some ETH. You can leave them around, but they don't bring yield to you. You can participate in ETH staking but it means you cannot use your ETH anymore. However, you can deposit ETH in a vault smart contract that participates in the ETH staking. As the representation of your assets in the vault smart contract, you get the tokens from the smart contract. Let's say the name of the token is VETH. Then, you can, for practical purposes, use VETH as the ETH replacement. If you don't use it, you still get yield from your deposited ETH in the vault smart contract.

Another operation from the vault smart contract is assets management. There are users who deposit assets into the vault. However, some managers or operators need to manage the assets in the vault. Otherwise, why do you put assets in the vault? The manager can study the economy and decide where to invest the assets. If you do not like assets being managed by a manager, you can even decide by voting. So, there are some governance actions in the vault.

This brings us to another concept: the **rebalancing act**. Here's an example: at the beginning of the year, the manager of the vault decided to invest assets in investment A, but at the end of the year, investment B looks brighter. The manager can decide to move assets from investment A to investment B.

As in the case of ERC-20 tokens and ERC-721 NFT standards, there is also a standard for vault smart contracts.

The ERC-4626 standard

In the beginning, people created vault smart contracts. These smart contracts at the time have different functions and mechanisms. For illustration, in a vault smart contract, the function to deposit assets is named `put_asset` but in another vault smart contract, the function to deposit assets can be named differently such as `deposit`. This makes it hard for composability. Suppose you want to deposit assets programmatically to many vault smart contracts – you have to find out what the method name is one by one.

However, what if every vault smart contract adheres to a standard? They agree that for a method that deposits assets, the method should be named `deposit`. That way, you don't have to worry about this naming issue when depositing assets into many vault smart contracts.

Fortunately, there is the ERC-4626 tokenized vault standard. The full specification of the standard can be read at this URL: `https://eips.ethereum.org/EIPS/eip-4626`.

There are events and function signatures you need to implement so your smart contract follows the ERC-4626 standard.

First, you must implement two events:

```
event Deposit(address indexed sender, address indexed owner, uint256
assets, uint256 shares)
event Withdraw(address indexed sender, address indexed receiver,
address indexed owner, uint256 assets, uint256 shares)
```

The `Deposit` event is triggered when a user deposits assets into the vault. The `Withdraw` event is triggered when a user withdraws asserts from the vault.

Then, you must implement some of the function signatures:

```
function asset() public view returns (address)
function totalAssets() public view returns (uint256)
function convertToShares(uint256 assets) public view returns (uint256
shares)
function convertToAssets(uint256 shares) public view returns (uint256
assets)
function maxDeposit(address receiver) public view returns (uint256)
function previewDeposit(uint256 assets) public view returns (uint256)
function deposit(uint256 assets, address receiver) public returns
```

```
(uint256 shares)
function maxMint(address receiver) public view returns (uint256)
function previewMint(uint256 shares) public view returns (uint256)
function mint(uint256 shares, address receiver) public returns
(uint256 assets)
function maxWithdraw(address owner) public view returns (uint256)
function previewWithdraw(uint256 assets) public view returns (uint256)
function withdraw(uint256 assets, address receiver, address owner)
public returns (uint256 shares)
function maxRedeem(address owner) public view returns (uint256)
function previewRedeem(uint256 shares) public view returns (uint256)
function redeem(uint256 shares, address receiver, address owner)
public returns (uint256 assets)
function totalSupply() public view returns (uint256)
function balanceOf(address owner) public view returns (uint256)
```

Let's discuss these points one by one:

- The `asset` function returns the address of the underlying asset of the vault. So, each vault can accept one asset. The asset is the ERC-20 token, such as USDT, UNI, USDC, and PEPE. You can set the asset in the initialization function for the vault or hardcode it in the smart contract.
- The `totalAssets` function returns the number of the underlying assets held by the vault. Suppose that User A has deposited 10 USD and User B has deposited 100 USD since the deployment of the vault; then, the total underlying asset will be 110 USD.
- The `convertToShares` function returns the number of shares based on the asset a user deposits. Suppose User A has deposited 10 USD since the beginning of the deployment of the vault; then, User A's total shares are 10 shares, assuming a 1:1 price between assets and shares. However, if User B deposits 10 USD, and you want to know how many shares 10 USD is equivalent to, then you can invoke this function and you will get 10 shares for 10 USD. However, if the price between assets and shares is 1:2 (1 USD gets 2 shares), then you will get a different answer, which is 20 shares.
- The `convertToAssets` function does the reverse thing of what `convertToShares` does. It returns the number of assets based on the shares you have. So, for the 20 shares you have, this function will return 10 USD with the 1:2 price between assets and shares.
- The `maxDeposit` function returns the maximum deposit a user can make. This is to prevent a certain user from depositing too many assets. In a vault smart contract, a user who has the most shares can dictate how asset management should be done. However, the user could have malicious intent.
- The `previewDeposit` function returns the shares a user has if they deposit a certain amount of assets. It simulates how many shares a user will get from a hypothetical deposit.

- The `deposit` function is used by a user to deposit assets into the vault. A user can then grant shares to a receiver with the second argument. The receiver can be themselves or other users.
- The `maxMint` function is used to prevent minting too many shares for a certain user. Again, since shares can be used for governance, this is to prevent power abuse.
- The `previewMint` function is to simulate the number of tokens that can be minted based on shares.
- The `mint` function is to create new shares for a recipient. You need assets to be able to mint new shares. If you want to mint 10 shares, then you need 10 USD (if the price is 1:1).
- The `maxWithdraw` function is to limit how many assets you can withdraw from the vault with a single withdrawal. It's similar to the concept of a withdrawal limit enforced by banks.
- The `previewWithdraw` function is to simulate how many shares a user would let go from a withdrawal of a number of assets.
- The `withdraw` function is to withdraw assets and burn shares. The assets will later be given to the recipient in the argument.
- The `maxRedeem` function is to limit how many shares a user can redeem in a single call.
- The `previewRedeem` function is to simulate how many assets a user would get from this redemption call.
- The `redeem` function is to redeem shares.
- The `totalSupply` function returns the total shares in the vault.
- The `balanceOf` function returns the number of shares a user has in the vault.
- The `mint` and `deposit` functions have the same purpose, but the mechanism is a bit different. They accept different arguments. The `mint` function accepts the `shares` argument to be minted. The `deposit` function accepts the `assets` argument to be sent into the vault. The end goal is the same.
- The `withdraw` and `redeem` functions have the same purpose, but the mechanism is a bit different. They accept different arguments. The `withdraw` function accepts the assets argument to be withdrawn. The `redeem` function accepts the `shares` argument to be burnt. The end goal is the same.

After learning the standard, let's write the vault smart contract with Vyper. The full smart contract code can be found at this URL: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_15/vault/contracts/Vault.vy`.

You can download the `Vault.vy` file from the URL and put it inside the `contracts` folder. You can also write `Vault.vy` from scratch. Create a new file named `Vault.vy` inside the `contracts` folder and add the following code:

```
#pragma version ^0.3.0

# From https://github.com/fubuloubu/ERC4626/blob/main/contracts/
VyperVault.vy

from vyper.interfaces import ERC20
from vyper.interfaces import ERC4626

implements: ERC20
implements: ERC4626
```

The `pragma` line tells us to compile this code with a Vyper version of at least 0.3. You imported two interfaces, which are ERC-20 and ERC-4626. Then, you declare in the smart contract to implement both interfaces. However, why must we adhere to the ERC-20 standard? We must because the vault smart contract has shares. The shares are the ERC-20 token. It means the vault smart contract must be able to do what the ERC-20 token smart contract can do.

After declaring that the smart contract follows the ERC-20 standard, you add the ERC-20 standard methods by adding the following code:

```
totalSupply: public(uint256)
balanceOf: public(HashMap[address, uint256])
allowance: public(HashMap[address, HashMap[address, uint256]])

NAME: constant(String[11]) = "Hello Vault"
SYMBOL: constant(String[6]) = "vHELLO"
DECIMALS: constant(uint8) = 18

event Transfer:
    sender: indexed(address)
    receiver: indexed(address)
    amount: uint256

event Approval:
    owner: indexed(address)
    spender: indexed(address)
    allowance: uint256
```

These functions and events are part of the ERC-20 standard.

Then, let's add events and a standard function from the ERC-4626 standard by adding the following code:

```
asset: public(ERC20)

event Deposit:
    depositor: indexed(address)
    receiver: indexed(address)
    assets: uint256
    shares: uint256

event Withdraw:
    withdrawer: indexed(address)
    receiver: indexed(address)
    owner: indexed(address)
    assets: uint256
    shares: uint256
```

The `asset` variable is the address of the underlying asset in this vault smart contract.

The `Deposit` event is triggered when a user calls the `deposit` function or the `mint` function. The `depositor` argument is the one who triggers the event. The `receiver` event is the one who receives the shares. The `assets` argument is the number of assets being deposited. The `shares` argument is the number of shares being minted.

The `Withdraw` event is triggered when a user calls the `redeem` function or the `withdraw` function. The `withdrawer` argument is the one that calls the `withdraw` function or the `redeem` function. The `receiver` argument is the one who will get the assets back from the shares being burnt. The `owner` argument is the one who owns the shares being burnt. The `assets` argument is the number of assets being withdrawn. The `shares` argument is the number of shares being burnt.

Then, let's add the initialization function by adding the following code:

```
@external
def __init__(asset: ERC20):
    self.asset = asset
```

In the initialization function, you initialize the underlying asset. You don't have to do it this way. You can hardcode it, for example.

Then, you need to add other standard ERC-20 functions by adding the following code:

```
@view
@external
def name() -> String[11]:
    return NAME
```

```
@view
@external
def symbol() -> String[6]:
    return SYMBOL


@view
@external
def decimals() -> uint8:
    return DECIMALS
```

These three functions return the name, the symbol, and the decimals of the shares token of the vault smart contract.

Then, let's add the remaining methods of the ERC-20 standard by adding the following code:

```
@external
def transfer(receiver: address, amount: uint256) -> bool:
    self.balanceOf[msg.sender] -= amount
    self.balanceOf[receiver] += amount
    log Transfer(msg.sender, receiver, amount)
    return True


@external
def approve(spender: address, amount: uint256) -> bool:
    self.allowance[msg.sender][spender] = amount
    log Approval(msg.sender, spender, amount)
    return True


@external
def transferFrom(sender: address, receiver: address, amount: uint256)
-> bool:
    self.allowance[sender][msg.sender] -= amount
    self.balanceOf[sender] -= amount
    self.balanceOf[receiver] += amount
    log Transfer(sender, receiver, amount)
    return True
```

The `transfer` function is to transfer assets from one account to another account. The `approve` function is to give approval to an account so the account can transfer assets on behalf of another account with the `transferFrom` function.

It's useful to know the total assets that the vault has. Let's add the code to achieve this goal:

```
@view
@external
def totalAssets() -> uint256:
    return self.asset.balanceOf(self)
```

`totalAssets` checks the balance of the underlying asset from the underlying token smart contract.

Then, the next function you want to add is the `convertToShares` function. Do that by adding the following code:

```
@view
@internal
def _convertToShares(assetAmount: uint256) -> uint256:
    totalSupply: uint256 = self.totalSupply
    totalAssets: uint256 = self.asset.balanceOf(self)
    if totalAssets == 0 or totalSupply == 0:
        return assetAmount  # 1:1 price

    return assetAmount * totalSupply / totalAssets


@view
@external
def convertToShares(assetAmount: uint256) -> uint256:
    return self._convertToShares(assetAmount)
```

The `totalSupply` variable refers to the total number of shares. The `totalAssets` variable refers to the total underlying assets in the vault. The `assetAmount` variable is the number of assets you want to convert to the number of shares. In this smart contract, the price is 1:1, so one asset will be converted to one share. For example, if you deposit 1 USDT, you will get 1 vUSDT, assuming vUSDT is the name of the shares token in the vault and USDT is the underlying asset.

Then you need to add the `convertToAssets` function by adding the following code:

```
@view
@internal
def _convertToAssets(shareAmount: uint256) -> uint256:
    totalSupply: uint256 = self.totalSupply
    if totalSupply == 0:
        return 0

    return shareAmount * self.asset.balanceOf(self) / totalSupply
```

```
@view
@external
def convertToAssets(shareAmount: uint256) -> uint256:
    return self._convertToAssets(shareAmount)
```

This function converts the number of shares to the number of assets. It does the reverse thing of what `convertToAssets` does. Because the price is 1:1, 1 share will be converted to 1 asset.

Then, you want to add the `maxDeposit` function by adding the following code:

```
@view
@external
def maxDeposit(owner: address) -> uint256:
    return MAX_UINT256
```

In this case, you don't want to limit the amount of assets in one single deposit.

Then, you want to add a function to simulate the effect of a deposit action by adding the following code:

```
@view
@external
def previewDeposit(assets: uint256) -> uint256:
    return self._convertToShares(assets)
```

It uses the internal `_convertToShares` function that the `convertToShares` function uses. So, in this case, it's not really different from the `convertToShares` function but in real life, you want to add complex logic here, such as subtracting assets with a fee.

Then, you want to add the `deposit` function by adding the following code:

```
@external
def deposit(assets: uint256, receiver: address=msg.sender) -> uint256:
    shares: uint256 = self._convertToShares(assets)
    self.asset.transferFrom(msg.sender, self, assets)

    self.totalSupply += shares
    self.balanceOf[receiver] += shares
    log Transfer(empty(address), receiver, shares)
    log Deposit(msg.sender, receiver, assets, shares)
    return shares
```

In this function, you transfer the asset from the sender to the vault smart contract. Then, you increase the total supply of the shares. You credit this increase in the receiver balance. Then, you log the `Transfer` event and the `Deposit` event. Lastly, you return the amount of shares this function creates.

Then, you need to add the `maxMint` function to limit the minting action by adding the following code:

```
@view
@external
def maxMint(owner: address) -> uint256:
    return MAX_UINT256
```

In this case, you don't limit the maximum number of shares a user can mint.

Then, you want to simulate the effect of minting by adding the following code:

```
@view
@external
def previewMint(shares: uint256) -> uint256:
    assets: uint256 = self._convertToAssets(shares)

    if assets == 0 and self.asset.balanceOf(self) == 0:
        return shares

    return assets
```

This is similar to the `convertToAssets` function because we're using the 1:1 price. However, in real life, you may want to introduce complex logic, such as fees.

Then, you want to create the `mint` function by adding the following code:

```
@external
def mint(shares: uint256, receiver: address=msg.sender) -> uint256:
    assets: uint256 = self._convertToAssets(shares)

    if assets == 0 and self.asset.balanceOf(self) == 0:
        assets = shares

    self.asset.transferFrom(msg.sender, self, assets)

    self.totalSupply += shares
    self.balanceOf[receiver] += shares
    log Transfer(empty(address), receiver, shares)
    log Deposit(msg.sender, receiver, assets, shares)
    return assets
```

The logic of the `mint` function is similar to the `deposit` function but the argument is the number of shares. So, you need to convert the shares to the assets first before doing a similar thing in the `deposit` function.

After storing assets, you need to create functions to withdraw assets from the vault. Let's start by adding the `maxWithdraw` function:

```
@view
@external
def maxWithdraw(owner: address) -> uint256:
    return MAX_UINT256
```

Here, you don't limit the number of assets a user can withdraw in a single act.

Then, you want to create a function to simulate the effect of withdrawal by adding the following code:

```
@view
@external
def previewWithdraw(assets: uint256) -> uint256:
    shares: uint256 = self._convertToShares(assets)

    if shares == assets and self.totalSupply == 0:
        return 0

    return shares
```

The `previewWithdraw` function is the same as the `convertToShares` function because we're using the 1:1 price.

Then, this is the `withdraw` function you need to add:

```
@external
def withdraw(assets: uint256, receiver: address=msg.sender, owner:
address=msg.sender) -> uint256:
    shares: uint256 = self._convertToShares(assets)

    if shares == assets and self.totalSupply == 0:
        raise

    if owner != msg.sender:
        self.allowance[owner][msg.sender] -= shares

    self.totalSupply -= shares
    self.balanceOf[owner] -= shares

    self.asset.transfer(receiver, assets)
    log Transfer(owner, empty(address), shares)
    log Withdraw(msg.sender, receiver, owner, assets, shares)
    return shares
```

When you withdraw, you decrease the total supply of shares, and you debit it to the balance of the owner of the shares. If the sender is not the owner, you must ensure there is allowance for the sender on the owner's behalf. After that, you transfer the asset from the vault to the receiver. You log the `Transfer` event and the `Withdraw` event. Lastly, you return the shares being burnt from the withdrawal action.

After withdrawal, you want to add some `redeem`-related functions. First, you add the `maxRedeem` function by adding the following code:

```
@view
@external
def maxRedeem(owner: address) -> uint256:
    return MAX_UINT256
```

Then, you want to simulate the `redeem` action by adding the following code:

```
@view
@external
def previewRedeem(shares: uint256) -> uint256:
    return self._convertToAssets(shares)
```

The `previewRedeem` function is the same as the `convertToAssets` function because we don't have complex logic such as fees.

Lastly, you need to add the `redeem` function by adding the following code:

```
@external
def redeem(shares: uint256, receiver: address=msg.sender, owner:
address=msg.sender) -> uint256:
    if owner != msg.sender:
        self.allowance[owner][msg.sender] -= shares
    assets: uint256 = self._convertToAssets(shares)
    self.totalSupply -= shares
    self.balanceOf[owner] -= shares

    self.asset.transfer(receiver, assets)
    log Transfer(owner, empty(address), shares)
    log Withdraw(msg.sender, receiver, owner, assets, shares)
    return assets
```

The logic of the `redeem` function is the same as the logic of the `withdraw` function but it accepts shares, not assets as the argument.

Now, you've written the full vault smart contract. You can compile it by running the following command:

```
(.venv) $ ape compile
```

After writing the vault smart contract, you want to create a lending smart contract.

Lending application

As discussed previously, lending can be combined with a vault, meaning the pooled resources can be used for lending.

Everyone knows what lending is. You lend money to your friend, or in other words, your friend borrows money from you. You charge interest to this lending process. After a certain period of time, your friend pays back the money to you with the interest.

You may lend money to your friend out of kindness, but banks lend money to companies for profit. Companies borrow money so they can expand their businesses.

You can also create a lending application on top of the blockchain. So, users can borrow assets from smart contracts. Later, borrowers need to return the assets with the interest. If not, their collateral will be liquidated.

You can visualize a lending protocol in the following image where **User A** provides **4000 USDT** as an asset in the lending protocol so **User B** can borrow **1000 USDT**.

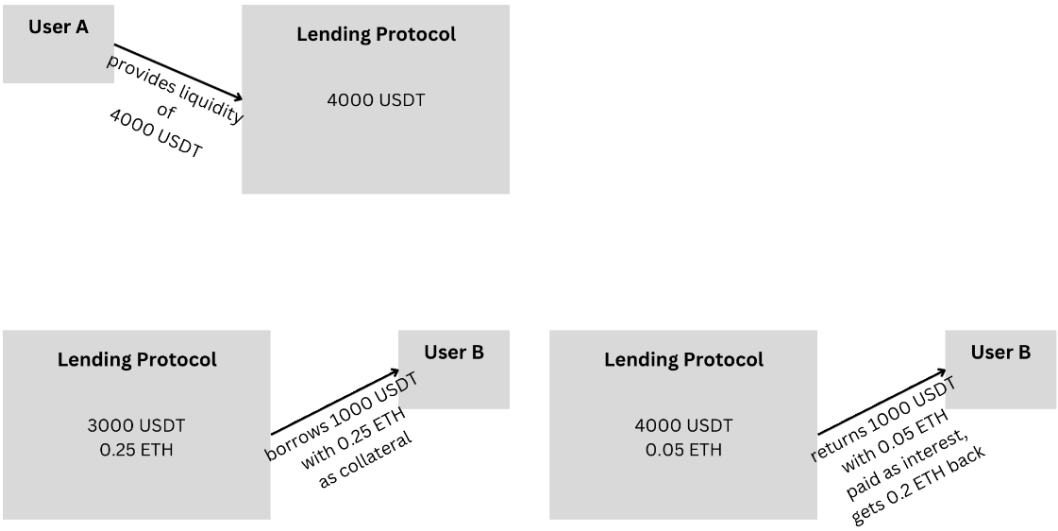


Figure 15.2: A lending protocol

Let's create a simple lending smart contract. However, this time, let's just create a lending smart contract without a vault attached. That way, you can understand the concept of lending in the smart contract easily.

You can start the journey by creating a new file named `Lending.vy` inside the `contracts` folder. Then, you can add the following code:

```
#pragma version ^0.3.0

event Lend:
    _borrower: indexed(address)
    _amount: indexed(uint256)

event Repay:
    _borrower: indexed(address)
    _amount: indexed(uint256)
```

The `pragma` line tells us to compile the smart contract file with a Vyper compiler version of at least 0.3. Then, there are two events: `Lend` and `Repay`. The `Lend` event will be triggered when a lending process or a borrowing process takes place. The `Repay` event will be triggered when a borrower repays their loan. Both events use two arguments: the first is the borrower's address and the second is the loan amount.

After these events, you need to add the ERC-20 interface to the smart contract because this is the asset a borrower wants to borrow and the smart contract wants to lend. Let's add the following code:

```
interface ERC20_Interface:
    def transfer(_recipient: address, _amount: uint256): nonpayable
    def transferFrom(_sender: address, _recipient: address, _amount:
uint256): nonpayable
```

In the lending smart contract, you only want to use two functions from the ERC-20 interface, the `transfer` function and the `transferFrom` function.

Then, you need to add these state variables:

```
token: public(address)
loan_amount: public(uint256)
collateral: public(uint256)
borrower: public(address)
loan_taken: public(bool)
interest: public(uint256)
owner: public(address)
```

The `token` variable refers to the asset that the smart contract will lend to a borrower. The `loan_amount` variable is the number of assets the smart contract will lend. The `collateral` variable is the number of the ETH collateral that must be sent to the smart contract to borrow the asset. The `borrower` variable is the account the smart contract permits to borrow from this smart contract. The `loan_taken` variable is the status of the lending in the smart contract. If it's true, then the lending has taken place. The `interest` variable is the interest percentage that the borrower needs to pay after repaying the loan. The `owner` variable refers to the owner or the deployer of the smart contract.

Let's create the initialization function to initialize these state variables:

```
@external
def __init__(_borrower: address,
             _token: address,
             _loan_amount: uint256,
             _collateral: uint256,
             _interest: uint256):
    self.token = _token
    self.loan_amount = _loan_amount
    self.collateral = _collateral
    self.borrower = _borrower
    self.interest = _interest
    self.owner = msg.sender
```

You initialize the state variables from the arguments of the function, but you initialize the owner variable with `msg.sender` or the account that executes the initialization function.

The amount of the loan, the interest, and the borrower are set from the beginning to make the smart contract simple. So, after the smart contract is deployed, only the specified borrower can borrow the specific asset with a specific amount. However, in real life, you need to be more flexible with the lending smart contract.

Then, let's add the function to borrow assets by adding the following code:

```
@external
@payable
def borrow():
    assert msg.sender == self.borrower, "Only the borrower can borrow
the asset"
    assert msg.value == self.collateral, "Collateral is not enough"
    ERC20_Interface(self.token).transfer(msg.sender, self.loan_amount)
    self.loan_taken = True

    log Lend(msg.sender, self.loan_amount)
```

First, you make sure only the borrower calls this function. Then, you make sure the amount of ETH sent to the smart contract is as high as the `collateral` variable. After this, the smart contract sends a number of assets as big as the `loan_amount` variable. Then, you set the `loan_taken` variable to be true, indicating the lending process has taken place. Lastly, you trigger the `Lend` event.

This is collateralized lending, meaning to lend you need collateral. Outside blockchain, an example would be to borrow $100,000 from a bank – you need collateral, such as a car or a house. If you fail to repay the loan with the interest, your collateral (a house or a car) will be confiscated.

The same principle applies to the `borrow` function. Say the asset to be borrowed is USDT, the loan amount is 2000, and the collateral is 1 ETH. If a borrower wants to borrow 2000 USDT, they need 1 ETH as collateral. If they fail to repay the loan, which is 2000 USDT, their 1 ETH can be confiscated.

There is, of course, a lending process without collateral. If you fail to pay the loan, there is no collateral to be confiscated. However, your name will be put onto a blacklist, meaning you cannot get a loan anymore from the lender.

You can also apply that to the smart contract if you want to. You can even check their historical transactions to make sure they are eligible to borrow assets or not. However, for this tutorial, collateralized lending is simpler. So, we use that.

Now, let's add a function to a function to repay the loan by adding the following code:

```
@external
def repay():
    assert msg.sender == self.borrower, "Only the borrower can repay
the loan"
    assert self.loan_taken == True, "Loan has not been taken"

    cut: uint256 = self.interest * self.collateral / 100
    payback: uint256 = self.collateral - cut

    ERC20_Interface(self.token).transferFrom(msg.sender, self, self.
loan_amount)
    send(msg.sender, payback)

    self.loan_taken = False

    log Repay(msg.sender, self.loan_amount)
```

In the `repay` function, some validations validate only the borrower calling this function and the loan is taken already. Then, you calculate the interest amount and set it to the `cut` variable. The interest will be applied to the collateral, not the borrowed asset. So, you get the revenue from the collateral, not the borrowed asset. Then, you make sure you get the borrowed asset from the borrower, and then you send back the borrower's collateral minus the interest. After that, you set it that the lending process has ended. Lastly, you log the `Repay` event.

Don't forget to add a function to withdraw the revenue or the interest you get from the lending:

```
@external
def withdraw_eth():
    assert msg.sender == self.owner, "Only the owner can withdraw ETH"
    assert self.loan_taken == False, "Cannot withdraw while loan is
active"
    send(msg.sender, self.balance)
```

You make sure the caller of the function is the owner. Then, you make sure the loan has ended. Lastly, you transfer all ETH inside the smart contract to the owner.

You can compile the smart contract by executing the following command:

```
(.venv) $ ape compile
```

The lending smart contract is not yet complete. There is no function to confiscate the collateral in case the borrower fails to repay the loan. In this smart contract, we assume the borrower succeeds in fulfilling their duty. You can add one if you want to. You can add a time condition by checking the block time in the function. If it passes the time you set, then you can withdraw their collateral.

You can take a look at the test file of the lending smart contract to see how it works. The test file can be downloaded from this URL: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_15/vault/tests/test_lending.py`. Don't forget to check the fixture file as well. It can be downloaded from this URL: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_15/vault/tests/conftest.py`.

You can see that, before a borrower can borrow assets from the smart contract, someone needs to transfer the assets to the smart contract. Otherwise, where do the assets to be borrowed come from?

So, if you want to create a lending vault, you can combine these two smart contracts, the vault smart contract and the lending smart contract. In the vault smart contract, users deposit assets into the smart contract. That's where the assets to be borrowed come from. The lending vault smart contract can be downloaded at this URL: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_15/vault/contracts/LendingVault.vy`. Basically, the lending vault smart contract is the combination of both smart contracts with some adjustments.

You can visualize a lending vault where many users deposit assets to provide liquidity in the lending vault so other users can borrow assets in the following image.

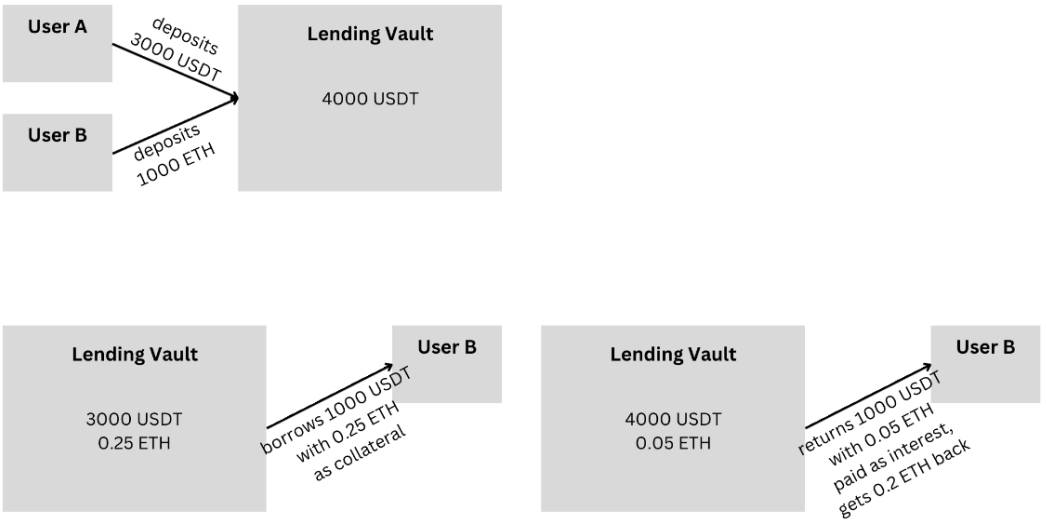


Figure 15.3: The lending vault

The lending vault is not yet complete. The revenue in ETH cannot be withdrawn yet. You cannot just use the withdrawal function from the lending smart contract because you have to share it with depositors. This is to incentivize people so they want to deposit assets in the vault. Remember that depositors get shares? The bigger the shares, the more profit they can withdraw.

Although the lending vault is not complete, this chapter has walked you through the steps of building the lending vault smart contract. This gives you a foundation on how you can expand your lending vault smart contract.

Most of the lending protocols that are popular on the blockchain use vaults, such as Aave (`https://aave.com`). Many users can provide assets to the vault so other users can borrow assets from the lending protocol using the vault. This is better because the liquidity becomes bigger.

Summary

In this chapter, you learned about the concept of vaults. The vault is where people deposit and withdraw assets. There are categories of vaults, such as lending vaults, liquidity vaults, and staking vaults. Then, you studied the ERC-4626 standard, which is the vault smart contract standard. You learned about all of the standard functions, including functions to deposit and withdraw assets. You also wrote the vault smart contract. Then, you learned about lending by writing the lending smart contract. You understood how the lending smart contract implements lending. Lastly, you saw how to combine the lending smart contract and vault smart contract into a lending vault smart contract. This chapter gave you skills in building two of the most important and popular protocols in the blockchain, which are the vault protocol and the lending protocol. In the next chapter, you'll learn how to write an automated market maker decentralized exchange smart contract.

16
Decentralized Exchange

In this chapter, you're going to learn about **decentralized exchanges** (**DEXs**), one of the most popular kinds of applications on the blockchain. Some famous DEXs are Uniswap, Sushi, Curve, and so on. In these exchanges, people trade or swap their ERC-20 tokens. These exchanges are decentralized, meaning it's not the same as stock trading applications, which are centralized. You'll also learn the algorithm that these exchanges use, which is **Automated Market Maker** (**AMM**). Then, you'll write a DEX smart contract using the AMM algorithm. This chapter focuses on DEXs that use the AMM algorithm, such as Uniswap. There are other DEXs that use different algorithms, such as order books and AMM, but with the StableSwap AMM algorithm used by Curve.

The following topics will be covered in this chapter:

- Exploring DEXs
- Understanding AMM
- Writing a DEX smart contract

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_16`.

Exploring DEXs

An exchange is a place where you can buy and sell stocks of companies. If you want to buy some shares of the Apple company, you can buy AAPL stocks on the exchange through applications such as Robinhood. Now, let us assume that you own a great company, and you want to expand it. You need to get capital from the public. In such a case, you can go public with an **initial public offering** (**IPO**). You can release some shares (for example, 20%) to an exchange, and people can buy the shares so that they can own some parts of your company.

This kind of exchange is centralized. There are some properties of a **centralized exchange** (**CEX**). A CEX is not up for 24 hours. Regular trading hours in the **New York Stock Exchange** (**NYSE**) and the **National Association of Securities Dealers Automated Quotations** (**Nasdaq**) are from 9:30 a.m. to 4 p.m. Normal trading hours in other countries are similar or not far off. No – you cannot trade stocks on holiday!

A CEX has also geographic limitations, meaning Indonesian people cannot buy AAPL stocks on Nasdaq because they are not Americans. Americans also cannot buy stocks of Indonesian companies. Yes – there is a workaround, such as a company buying and selling stocks of American companies on behalf of Indonesian people. But that is an exception. Indonesian people cannot easily buy stocks of foreign companies in other countries outside the US, such as India, Japan, South Korea, and so on.

There is also another risk in a CEX, such as intervention from the exchange itself. An example from real life is Robinhood restricting buying shares of GameStop because of volatile stocks and regulatory requirements.

CEXs are not just for buying and selling stocks. There are CEXs where people can trade cryptocurrencies such as Binance, Coinbase, or Kraken. There are some differences between stock exchanges and cryptocurrency exchanges. You can trade cryptocurrencies 24 hours on a cryptocurrency exchange.

However, some situations that happen on a centralized stock exchange still apply to a centralized cryptocurrency exchange. An exchange can manipulate trading activities. Suppose you bought cryptocurrency XYZ yesterday at the price of 1 USD. Then, today, the price increased to 10 USD. You wanted to sell it to get some profits. But alas, for some unknown reason, you couldn't sell the cryptocurrency. The exchange could just explain that there are technical issues that prevent people from selling cryptocurrency. After a few hours, you could sell it again, but the price has dropped to 2 USD. What if the exchange prevented you from selling and used the opportunity to sell the cryptocurrency that the owner of the exchange owned? It's hard to prove fraud from the user side.

Other than that, another big risk from using a centralized cryptocurrency exchange is the funds that you deposit into the exchange could be embezzled by employees or owners of the exchange, just like what happened on the FTX exchange.

What if you could prevent or at least mitigate some risks that are present in a CEX with blockchain technology?

That's where a DEX comes in. A DEX is basically just a smart contract. Just as with the non-fungible token (NFT) marketplace, you can create an exchange where people trade cryptocurrencies in the form of smart contracts.

As you've seen in previous chapters, a smart contract brings added value to the exchange. You get transparency. All transactions are there for you to digest in the blockchain. In a CEX, you have to pay if you want to get historical transactions.

The longevity of smart contracts is underrated. A CEX can be shut down, just like FTX was shut down. But to shut down DEX smart contracts, you need to shut down the blockchain itself (or Ethereum, in this case).

To prevent intervention from the owner, you can write smart contracts such that even the owner of the smart contract cannot intercept any trade activity. That way, nobody can stop any transaction in the exchange if you wish.

Just as stated in previous chapters, you can, of course, write DEX smart contracts such that the owner can censor or intercept transactions. But the rules are transparent. Users can choose whether to participate or not in the DEX you created.

A DEX, of course, runs 24 hours a day every day, just like any other smart contract. It only stops running when the blockchain (or Ethereum, in this case) is down.

Unlike in a centralized stock exchange, if you want to go public with your company, you have to fulfill the IPO requirements. If you want to list a cryptocurrency that you've created in CEXs, there are some fees that you need to pay unless your cryptocurrency is very popular. But in a DEX, you can just list your cryptocurrency freely, and you don't need to ask permission from anyone. The fees that you need to pay are fees for deploying smart contracts in Ethereum. DEX smart contracts can also charge fees for listing cryptocurrency, but that's up to the owner of the smart contracts. Some of the popular DEX smart contracts don't charge for listing cryptocurrency.

> **Note**
> DEX smart contracts also don't have geographic restrictions. People from all over the world can use DEX smart contracts as long as they can access Ethereum.

Some of the most popular DEX smart contracts are Uniswap, PancakeSwap, Curve, and Sushi.

Now that you understand DEX, it's time to understand the very popular algorithm that runs in the most popular DEX: AMM!

Understanding AMM

If you've traded stocks in any exchange, you should be familiar with the list of prices of a stock that traders are willing to sell and another list of prices of a stock that traders are willing to buy. To give an example, let's consider the XYZ stock. Four people want to sell the XYZ stock at different prices. Then, three people want to buy the XYZ stock at different prices as well. Let's make a list of the prices:

- Trader A wants to sell 10 XYZ stocks at $11
- Trader B wants to sell 12 XYZ stocks at $12
- Trader C wants to sell 11 XYZ stocks at $13

- Trader D wants to sell 18 XYZ stocks at $14
- Trader E wants to buy 15 XYZ stocks at $9
- Trader F wants to buy 13 XYZ stocks at $8
- Trader G wants to buy 18 XYZ stocks at $4

A potential buyer can buy XYZ stocks right now if they're willing to spend $11 at that time. There is no other lower price. Also, they can only buy 10 XYZ stocks. If they want to buy 15 XYZ stocks, for example, they need to buy 10 XYZ stocks at the price of $11 and 3 XYZ stocks at the price of $12. This happens because the sellers are only willing to sell XYZ stocks at those prices. If the buyer wants to buy XYZ stocks at lower prices, they need to wait until willing sellers at lower prices show up.

In fact, there is another side of the coin. There are buyers who want to buy XYZ stocks at certain prices, which are $9, $8, and $4. A potential seller who wants to sell their XYZ stocks right now can do so if they want to accept the price of $9 because that is the highest price that buyers want to accept to buy XYZ stocks. If a seller wants to sell 18 XYZ stocks, the seller needs to accept two prices, $9 and $8, because there isn't enough demand for the price of $9.

This is what happens in a stock exchange and centralized cryptocurrency exchanges. Buyers and sellers are free to post their offers whether they want to sell stocks or buy stocks. Offers are recorded in the exchange for all traders to see. Potential buyers and sellers (or traders, in short) can search for offers they like.

This way of organizing trade is called an **order book**. People's orders are recorded on a list and then put inside a book. People can see all orders and consume orders if they wish.

A stock exchange and a CEX use this algorithm in their trade.

Market makers

But what if there aren't enough offers on the exchange? Suppose you want to buy 100 XYZ stocks using the previous example; there isn't enough supply for your demand. To fulfill your demand, there needs to be enough XYZ stocks that are offered to be sold.

So, you need to persuade sellers to offer a lot of XYZ stocks at certain prices so that potential buyers can buy the stocks. Looking at the other side of the coin, you also need a lot of demand for XYZ stocks so that potential sellers can sell the stocks as well.

If there are no offers for buying and selling the XYZ stocks, potential buyers and potential sellers cannot do their transactions. A potential buyer and a potential seller can come to the exchange, and if their expected prices match, then the transaction can happen. However, the probability is low, and the process is not efficient. What if the potential buyer wants to buy XYZ stocks at 1 p.m. today but the potential seller wants to sell XYZ stocks at 3 p.m. tomorrow? The potential buyer must wait for a day to buy XYZ stocks.

To solve that problem, an exchange can ask people to become sellers and buyers. In other words, the exchange invites people to supply XYZ stocks or inventory. Why would people do that? The exchange can give them money. Every transaction in the exchange has a fee attached. The more the transactions, the better the exchange is. So, it's a win-win situation for the exchange and inventory suppliers. Another name for inventory suppliers is market makers.

Market makers don't have to get money from the exchange. They can make money on their own. They can buy XYZ stocks at the price of $10, then sell XYZ stocks at the price of $10.5 and pocket the difference.

But how do they know at what price they should sell or buy? They gauge the level of demand and supply. If there is more demand than supply of XYZ stocks, the market maker will increase the price. If the reverse thing happens, the market maker will decrease the price. It looks easy, but the market maker needs to be able to create a mathematical model. They also need to know the order flow and many things.

So, that's how an order book exchange works. The exchange needs market makers.

But you're not going to write an order book exchange smart contract, not because it's wrong. There's nothing wrong with it. There are order book exchange smart contracts.

If you want to create one, you can look into the NFT marketplace smart contract in *Chapter 14*. It's true that the NFT marketplace is not an exchange. But they're both markets. In the NFT marketplace, a user can propose a price for an NFT they own if they want to sell the NFT. Another user also can propose a price for an NFT they'd like to buy. As you know, these proposed prices can be recorded in a list, just like in an order book in an exchange.

The difference is that an NFT is non-fungible or unique, while stocks are fungible. So, instead of proposing a price for an NFT, you need to devise a way to propose a price for a number of stocks. For example, user A may want to propose 2 ETH for the whole 32 items of cryptocurrency XYZ. The buyer can take the offer but not fully. So, the buyer could pay 1 ETH to buy 16 items of cryptocurrency XYZ. Then, there are the remaining 16 items of cryptocurrency XYZ with the price of 1 ETH for the whole thing in the market. If you can make those adjustments, you can create an order book DEX smart contract.

The downside of using an order book is that the market makers need to be very active in maintaining the equilibrium between demand and supply. In smart contracts, you want to automate as much as possible. So, what if we can automate the maintenance of the price part? In fact, you can do that with AMM.

How AMM works

The most popular DEX application on Ethereum is Uniswap. Uniswap is a DEX application. The application is composed of many smart contracts. Uniswap doesn't use an order book. There are no buying or selling orders in Uniswap like you see in a centralized cryptocurrency exchange. But there is a price for the tokens you want to buy. How does Uniswap maintain the price of tokens? Uniswap uses AMM.

The basic premise of maintaining price is gauging demand and supply. If the demand becomes higher, the price should become higher as well. If the supply becomes higher, the price should become lower. That's the crude way of maintaining the price of any item.

In AMM, you use that premise as well. You can encode the equation into a smart contract. So, there's no need for other people to adjust the price of items or tokens. The smart contract handles it automatically.

One equation you can use to maintain the equilibrium between demand and supply is the following:

```
K = X * Y
```

`K` is a constant, meaning the value doesn't change at all. `*` is a times operation (2 * 8 = 16). `X` represents a reserve or supply of item A. `Y` represents a reserve or supply of item B.

That's it. You can put the equation into a smart contract, and it will handle the price of tokens automatically.

But how does a simple equation balance things out? Let's take a look at examples of how numbers fit into the equation. Suppose `K` is 12; then, the combination of `X` and `Y` would be the following:

```
K = X * Y

12 = 1 * 12

12 = 2 * 6

12 = 3 * 4

12 = 4 * 3

12 = 6 * 2

12 = 12 * 1
```

If `X` is 1, then `Y` should be 12, so `X` and `Y` would not violate the equation. If `X` is 2, then `Y` should be 6, and so on. Suppose `X` is US dollars, and `Y` is XYZ stocks. If there are 12 XYZ stocks, then the price of each XYZ stock is 1 US dollar (K = 1 * 12 and K is 12). If there are 6 XYZ stocks, then the price of each XYZ stock is 2 US dollars. The price increases because the supply becomes lower. Remember that if the supply becomes lower, the price should become higher to compensate for it. What if there is only 1 XYZ stock? Then the price of each XYZ stock is 12 US dollars! The rarer the item, the more expensive the item becomes.

It doesn't matter what the constant value is. The constant is there to keep the balance between two variables.

That way, you don't have to adjust the price manually in the DEX smart contract. If the supply of the tokens becomes low, then the price is increased automatically.

After understanding the equation, let's write an AMM DEX smart contract.

Writing a DEX smart contract

Let's start by creating an *Ape project*. You should follow these steps:

```
$ mkdir dex
$ cd dex
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install eth-ape'[recommended-plugins]'==0.7.23
(.venv) $ ape init
Please enter project name: DEX
SUCCESS: DEX is written in ape-config.yaml
```

The full code of the DEX smart contract can be found at this URL: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_16/dex/contracts/DEX.vy`. You can download it and place the smart contract file inside the `contracts` folder.

But let's create a smart contract from scratch. Create a new file named `DEX.vy` inside the `contracts` folder, then add the following code:

```
#pragma version ^0.3.0
# Based on https://jeiwan.net/posts/programming-defi-uniswap-1/

from vyper.interfaces import ERC20 as IERC20
```

The `pragma` line constrains the version of the Vyper compiler that can compile this smart contract file to be 0.3.x. Then, you import the `ERC20` interface because the DEX smart contract will interact with a cryptocurrency smart contract.

Then, you need to create custom interfaces for the DEX smart contract itself by adding the following code:

```
interface Exchange:
    def getReserve() -> uint256: view
    def getAmount(inputAmount: uint256,
                  inputReserve: uint256,
                  outputReserve: uint256) -> uint256: view
```

Later, you'll create a `getReserve` function and a `getAmount` function in the smart contract file. In other functions in the same smart contract, you'll need to call these external functions. To call external functions inside a function, you'll need to use the interface.

This DEX smart contract only handles one pair, which is ETH and an ERC-20 cryptocurrency. So, you need a way to keep track of the ERC-20 cryptocurrency smart contract by adding the following code:

```
tokenAddress: public(address)

@external
def __init__(_token: address):
    assert _token != ZERO_ADDRESS, "invalid token address"

    self.tokenAddress = _token
```

You can set the address of the ERC-20 cryptocurrency in the `tokenAddress` variable in the initialization function.

DEX needs an inventory so that people can buy this cryptocurrency. You should add a function to add liquidity by adding the following code:

```
@external
@payable
def addLiquidity(_tokenAmount: uint256):
    token: IERC20 = IERC20(self.tokenAddress)
    token.transferFrom(msg.sender, self, _tokenAmount)
```

To get an inventory of the cryptocurrency into the DEX, you just use the `transferFrom` function of the cryptocurrency smart contract to move some number of tokens from the sender to the DEX. You can use the ERC-20 interface to interact with the ERC-20 cryptocurrency smart contract. Notice that there's the `@payable` annotation in the function. It means that you need to send ETH along with the tokens. It's a pair of ETH and an ERC-20 cryptocurrency.

Once you get the tokens inside the DEX smart contract, you want to know how many tokens the DEX smart contract has. You should add a function to find out how big the reserve of the DEX smart contract is:

```
@external
@view
def getReserve() -> uint256:
    return IERC20(self.tokenAddress).balanceOf(self)
```

The function uses the `balanceOf` function of the ERC-20 smart contract to check how big the reserve of the smart contract is. Combined with the ETH balance of this smart contract, you can get the price of the token. Suppose in the smart contract balance, there are 80 ETH and 40 tokens. So, the price of 1 token is 2 ETH. The price of the token is basically the reserve of the token divided by the ETH balance.

With the `k = x * y` equation, the number of tokens you get if you spend some ETH does not exactly follow the equation. Suppose there are 80 ETH and 40 tokens. The price of 1 token is 2 ETH. If you spend 20 ETH, it doesn't mean you get 10 tokens because every ETH you spend will change the equilibrium.

If you spend 2 ETH and get 1 token, then after the transaction, the ETH balance is 78 ETH and the token reserve is 39 tokens. The danger of this situation is you can drain the DEX by buying the whole tokens with just 80 ETH. You want to create a situation where for every ETH spent on the DEX, the price of the tokens increases. To do that, you need to create a helper function to get the reduced amount of tokens if you spend ETH or vice versa by adding the following code:

```
@external
@view
def getAmount(inputAmount: uint256,
              inputReserve: uint256,
              outputReserve: uint256) -> uint256:
    assert inputReserve > 0 and outputReserve > 0, "invalid reserves"

    return (inputAmount * outputReserve) / (inputReserve +
inputAmount)
```

This is a general function that will be used when getting the token amount with ETH as input and getting the ETH amount with the token amount as input.

Let's add a function to get the token amount with the ETH amount as input by adding the following code:

```
@external
@view
def getTokenAmount(_ethSold: uint256) -> uint256:
    assert _ethSold > 0, "ethSold is too small"

    tokenReserve : uint256 = Exchange(self).getReserve()

    return Exchange(self).getAmount(_ethSold, self.balance,
tokenReserve)
```

The `getTokenAmount` function uses the `getAmount` function to calculate the token amount. The input amount is the amount of ETH. The input reserve is the ETH balance. `self.balance` refers to the amount of ETH that the DEX smart contract holds. The output reserve is the token reserve.

You also need another function that gets the ETH amount from the token amount. Let's add the following code:

```
@external
@view
def getEthAmount(_tokenSold: uint256) -> uint256:
    assert _tokenSold > 0, "tokenSold is too small"

    tokenReserve : uint256 = Exchange(self).getReserve()

    return Exchange(self).getAmount(_tokenSold, tokenReserve, self.
balance)
```

`getEthAmount` also uses the `getAmount` function, but it uses the token reserve as the input reserve and the ETH balance as the output reserve. The input itself is the amount of the token. Notice in this function, `self.balance` is sent as the third argument of `getAmount` while in `getTokenAmount`, `self.balance` is sent as the second argument.

Then, it's time to create a function to buy tokens with ETH. You can add the following code:

```
@external
@payable
def ethToTokenSwap(_minTokens: uint256):
    tokenReserve : uint256 = Exchange(self).getReserve()
    tokensBought : uint256 = Exchange(self).getAmount(msg.value,
                                        self.balance - msg.value,
                                        tokenReserve)

    assert tokensBought >= _minTokens, "insufficient output amount"

    IERC20(self.tokenAddress).transfer(msg.sender, tokensBought)
```

The `ethToTokenSwap` function accepts one argument. It's the minimum token that you want to accept if you buy tokens with ETH. The argument is important because you don't know exactly how many tokens you will get. Suppose the price of 1 token is 2 ETH and you want to buy 100 tokens with 200 ETH. But remember that each time you buy 1 token, the price of 1 token increases, so you will not get exactly 100 tokens with 200 ETH. Then, you need to remember other people could buy tokens before your purchase. After other people buy tokens, the price of the tokens increases as well. This is called slippage. The argument is to prevent the purchase from being completed if you think the price is becoming too expensive or the slippage is becoming too big.

To calculate how many tokens you get from the ETH you send to this function, you can use the `getAmount` function. You need to subtract the ETH balance from the ETH you send to this function. Then, you make sure the token amount you get is the same or bigger than the minimum token you send as the argument. Lastly, you transfer the tokens to the buyer with the `transfer` function of the ERC-20 cryptocurrency smart contract.

Then, you need to add a function to sell tokens by adding the following code:

```
@external
def tokenToEthSwap(_tokensSold: uint256, _minEth: uint256):
    tokenReserve: uint256 = Exchange(self).getReserve()
    ethBought: uint256 = Exchange(self).getAmount(_tokensSold,
                                    tokenReserve,
                                    self.balance)

    assert ethBought >= _minEth, "insufficient output amount"

    IERC20(self.tokenAddress).transferFrom(msg.sender, self, _
tokensSold)
    send(msg.sender, ethBought)
```

The `tokenToEthSwap` function accepts two arguments. The first argument is the number of tokens you want to sell. The second argument is the minimum amount of ETH you want to accept from this trade transaction. Slippage also happens when you want to sell tokens. You can't know for sure how many ETH you will get from selling these tokens. So, you can protect yourself with the minimum ETH amount argument in case you think the price of the token is becoming too low. Remember – you sell the tokens, so you prefer the price of tokens to be as high as possible.

The trade transaction of selling tokens is composed of two actions. The first step is to transfer tokens from the sender to the DEX. You achieve that by using the `transferFrom` function. The second step is to transfer ETH from the DEX to the sender. You do that by using the `send` function.

To see how the DEX smart contract works, you need an ERC-20 smart contract as well. Put one in the `contracts` folder so that the DEX smart contract can be initialized with the ERC-20 smart contract as well. You can use the `HelloToken.vy` file that you used in previous chapters. The `HelloToken.vy` file is available from this URL: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_16/dex/contracts/HelloToken.vy`. Don't forget to compile the smart contract as well.

You can compile both smart contracts by executing the following command:

```
(.venv) $ ape compile
```

To see how the DEX smart contract works, let's write some unit tests.

Unit tests

You should start writing a unit test by writing a fixture file. Create a new file inside the `tests` folder and name it `conftest.py`, then add the following code:

```
import pytest

@pytest.fixture
def deployer(accounts):
    return accounts[0]

@pytest.fixture
def token(deployer, project):
    return deployer.deploy(project.HelloToken, "Hello", "HEL", 3, 100)

@pytest.fixture
def exchange(deployer, project, token):
    return deployer.deploy(project.DEX, token.address)
```

The `deployer` function returns the deployer. The `token` function returns the deployed ERC-20 token smart contract. The `exchange` function returns the deployed DEX smart contract initialized with the address of the deployed ERC-20 token smart contract.

Then, let's create a test file named `test_DEX.py` inside the `tests` folder then add the following code:

```
import pytest


def test_addLiquidity(exchange, token, deployer):
    ETH = 100
    token.approve(exchange.address, 2000, sender=deployer)
    token_amount = 200
    exchange.addLiquidity(token_amount, value=ETH, sender=deployer)

    assert exchange.balance == ETH
    reserve = exchange.getReserve()
    assert reserve == token_amount
```

In the `test_addLiquidity` function, at first, you must approve the DEX smart contract to be able to transfer tokens on behalf of the user. Then, the user adds liquidity by calling the `addLiquidity` function of the DEX smart contract and sending a number of ETH to the DEX smart contract. To test whether the `addLiquidity` function works or not, you check the ETH balance of the DEX smart contract and the token reserve with the `getReserve` function of the DEX.

Then, let's test the `getTokenAmount` function by adding the following code:

```
def test_getTokenAmount(exchange, token, deployer):
    token_amount = 20000
    token.approve(exchange.address, token_amount, sender=deployer)
    ETH = 10000
    exchange.addLiquidity(token_amount, value=ETH, sender=deployer)

    tokensOut = exchange.getTokenAmount(10)
    assert tokensOut == 19

    tokensOut = exchange.getTokenAmount(100)
    assert tokensOut == 198

    tokensOut = exchange.getTokenAmount(1000)
    assert tokensOut == 1818

    tokensOut = exchange.getTokenAmount(5000)
    assert tokensOut == 6666
```

You can add liquidity first. There are `10000` wei (the smallest currency in ETH) and `20000` tokens. So, the price of a token is 0.5 wei, or 1 wei can buy 2 tokens. Then, you test the `getTokenAmount` function with the different numbers. If you send 10 wei, you'll get 19 tokens (not 20 tokens) because there is slippage. The bigger the number, the bigger the slippage.

Then, you need to create a test for the `getEthAmount` function by adding the following code:

```
def test_getETHAmount(exchange, token, deployer):
    token_amount = 20000
    token.approve(exchange.address, token_amount, sender=deployer)
    ETH = 10000
    exchange.addLiquidity(token_amount, value=ETH, sender=deployer)

    ethOut = exchange.getEthAmount(10)
    assert ethOut == 4

    ethOut = exchange.getEthAmount(100)
    assert ethOut == 49

    ethOut = exchange.getEthAmount(1000)
    assert ethOut == 476

    ethOut = exchange.getEthAmount(10000)
    assert ethOut == 3333
```

```
    ethOut = exchange.getEthAmount(15000)
    assert ethOut == 4285
```

As usual, you add liquidity first. Then, you test the `getEthAmount` function with different numbers. As a reminder, there are 10000 wei (the smallest currency in ETH) and 20000 tokens. So, the price of a token is 0.5 wei, or 1 wei can buy 2 tokens. Using 10 tokens as the argument for the `getEthAmount` function will get you 4 wei (not 5 wei) because there is slippage. The bigger the token amount argument, the bigger the slippage.

Now that you nailed getting amount functions, you need to add tests for swapping functions.

Let's start by adding a test for the `ethToTokenSwap` function:

```
def test_ethToTokenSwap(exchange, token, deployer, accounts):
    token_amount = 20000
    token.approve(exchange.address, token_amount, sender=deployer)
    ETH = 10000
    exchange.addLiquidity(token_amount, value=ETH, sender=deployer)

    exchange.ethToTokenSwap(15, value=10, sender=accounts[1])
    assert token.balanceOf(accounts[1]) == 19

    exchange.ethToTokenSwap(180, value=100, sender=accounts[2])
    assert token.balanceOf(accounts[2]) == 197

    exchange.ethToTokenSwap(1600, value=1000, sender=accounts[3])
    assert token.balanceOf(accounts[3]) == 1780
```

In this test function, after adding liquidity, you call the `ethToTokenSwap` function as the `accounts[1]` user. You send 10 wei with a minimum of 15 tokens. The result is you get 19 tokens (not 20 tokens) because there is slippage. Then, you call the `ethToTokenSwap` function again with different users and ETH payment amounts.

Then, let's add a test for the `tokenToEthSwap` function:

```
def test_tokenToEthSwap(exchange, token, deployer, accounts):
    token_amount = 20000
    token.approve(exchange.address, token_amount, sender=deployer)
    ETH = 10000
    exchange.addLiquidity(token_amount, value=ETH, sender=deployer)

    token.transfer(accounts[1], token_amount, sender=deployer)
    token.transfer(accounts[2], token_amount, sender=deployer)
    token.transfer(accounts[3], token_amount, sender=deployer)
```

```
    initial_balance = accounts[1].balance
    sold_token = 100
    ethAmount = 49
    token.approve(exchange.address, token_amount, sender=accounts[1])
    exchange.tokenToEthSwap(sold_token, 40, sender=accounts[1])
    assert token.balanceOf(accounts[1]) == token_amount - sold_token

    initial_balance = accounts[2].balance
    sold_token = 1000
    token.approve(exchange, sold_token, sender=accounts[2])
    exchange.tokenToEthSwap(sold_token, 400, sender=accounts[2])
    assert token.balanceOf(accounts[2]) == token_amount - sold_token
```

In this test function, you add liquidity and then transfer some tokens from the deployer balance to other users so that they can have tokens to sell. Then, before selling tokens, the buyers need to approve the DEX to spend tokens on behalf of the buyers. This way, the DEX can get the tokens from the buyers. After executing the `tokenToEthSwap` function, you check the number of tokens the buyers have has decreased.

You can execute the test file by executing the following command:

```
(.venv) $ py.test tests/test_DEX.py
```

This DEX smart contract is far from finished. It has many shortcomings. For one thing, you can send ETH to the DEX smart contract without going through the `addLiquidity` function. It means you disrupt the equilibrium of the `k = x * y` equation. To solve this problem, you can track the ETH that people send through the proper function.

Then, this DEX smart contract cannot handle the pair of ERC-20 cryptocurrencies. What if you want to pair USDT with PEPE? To add this feature, you need to replace ETH-related functions with ERC-20 cryptocurrency-related functions. Instead of sending ETH, you give approvals to the DEX so that the DEX can withdraw tokens from two cryptocurrency smart contracts on your behalf.

You may want to add a way to generate revenue. Every time there is a trade transaction, you can cut a fee and put it in the smart contract. You can share the revenue with liquidity providers.

Summary

In this chapter, you learned about what a DEX is and what values it brings to users. You saw that an exchange can have different ways to organize trade transactions using order books and market makers. Price adjustment becomes a hassle with this strategy. Then, to solve this, you learned about the AMM algorithm, where the price is adjusted automatically in the smart contract. You then built a smart contract that implemented a DEX using the AMM algorithm. The DEX smart contract could trade between ETH and an ERC-20 token smart contract. Finally, you tested the DEX smart contract and learned how it works. Learning to build a DEX smart contract can benefit readers in several ways. It provides a hands-on understanding of how **decentralized finance** (**DeFi**) applications work. Additionally, understanding the AMM algorithm and its implementation can enhance your knowledge of decentralized trading mechanisms. In the next chapter, which is the last chapter, you're going to write a token-gated application.

Part 7: Building a Full-Stack Web3 Application

In this section, you will delve into the realm of full-stack Web3 application development, where blockchain meets the web to create immersive and decentralized experiences. It serves as your guide to crafting a web application that leverages non-fungible tokens (NFTs) to grant access and unlock unique digital experiences.

This section has the following chapter:

- *Chapter 17*, *Token-Gated Applications*

17
Token-Gated Applications

In this chapter, you're going to learn how to write a token-gated application. First, you will study the concept of token-gated application. What does gating an application with tokens mean? After learning the concepts, you will see how to prove the ownership of tokens using a signature. You'll also use MetaMask, a wallet browser extension. With it, you can create a signature and interact with a token-gated application. This knowledge will be useful to link the blockchain and a web application.

The following topics will be covered in this chapter:

- Understanding token-gated applications
- Creating a signature
- Installing a wallet browser extension
- Deploying a token smart contract
- Writing a backend application
- Writing a frontend application

Technical requirement

The code for this chapter can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/tree/main/chapter_17`. The libraries and applications used in this chapter are as follows:

- Vyper version 0.3.10
- Ape Framework version 0.7.23
- FastAPI version 0.111
- Wagmi.sh version 2.9.8
- MetaMask version 11

- Node.js version 20
- Pnpm version 9

Understanding a token-gated application

To understand what a **token-gated application** is, consider this. There are two worlds: the *web3 world* and the *web2 world*. The web3 world consists of blockchains and smart contracts. The web2 world is web applications. The word "token" in "token-gated application" means ERC-20 tokens or NFT tokens. The word "gated" means limiting access (to the application). The word "application" means applications such as web applications. So, a token-gated application means a web application where access to some features is limited using NFT tokens or ERC-20 tokens. If this sounds too complicated, let me simplify it a bit. Let's assume that there is a web application that you want to use; perhaps it's a video streaming website, such as Disney+ or Netflix. A user wants to watch movies on this web application, but access is limited. Not everyone is able to watch movies from the get-go. You need to own certain ERC-20 tokens or NFT tokens and prove ownership to the web application so you can access the premium feature, which is watching movies.

So, you gate the web application with digital assets on the blockchain. What happens on the blockchain determines who gets access to the web application.

Why do we need to limit access to web applications with tokens on the blockchain? Why can't we limit access to web applications with traditional means such as username and password? What value do token-gated applications bring over normal applications?

To answer these questions, you need to go back to the value that the blockchain brings to users. In the course of this book, you've been told that a blockchain brings transparency, censorship resistance, flexibility, and many more.

Let's take one example of the benefits of using a blockchain as a platform. *What happens on the blockchain is transparent.* You can see who buys which NFT tokens and who owns which ERC-20 tokens. Nobody can prevent you from selling or buying NFT tokens in NFT marketplaces.

What if you want to bring that value outside blockchain? Would it be nice? Certainly, it would be nice if you could bring the transparency value from a blockchain to opaque web applications. That's where the token-gated application comes in. Of course, you cannot turn a web application into a 100% transparent application. But you can bring some transparency into a web application.

Let's come back to the video streaming web application again. The list of all movies in the web application is opaque from users' perspective. You cannot know the list unless you peek into the database, which you don't have access to. In a normal video streaming web application, people create accounts and buy subscriptions to access the premium feature, which is watching movies. But what if you can change the access from using accounts to using tokens on the blockchain?

Imagine that in order to watch movies, you must buy certain NFT tokens. This means there is some transparency to the data on who can watch movies in the streaming web application. It is possible to take a peek into the blockchain to see who owns these NFT tokens. That way, you can see the list of subscribers. But that's not so interesting. What good comes from knowing this information? Perhaps you want to satisfy your curiosity and find out whether your friend across the street bought a subscription to watch movies. However, the blockchain gives you more than just information. It also gives flexibility and control over your subscription. If you're bored with your subscription, you can sell or transfer your NFT tokens to other people. It is also possible to trade movie streaming web application subscriptions in the marketplace. You can even use the subscription as collateral for a loan!

Similarly, token-gated applications can also bring more value to the blockchain. While the blockchain is very big and there are a lot of applications on the blockchain, the real world is vast. Linking two worlds, the blockchain and the real world, would bring so much more value to both.

Imagine you want to build a private community. You could release an NFT collection that consists of 99 NFT tokens and then hold events or giveaways to NFT holders. Assets on the blockchain unlock opportunities in real life.

Now that you understand the concept of token-gated applications, a question appears. How does someone prove the ownership of tokens on the blockchain in web applications?

Creating a signature

Let us assume that you have built a token-gated movie streaming web application. To access the web application, a user needs to buy one NFT from an NFT collection that you released. One user buys an NFT so she can watch movies in your token-gated movie streaming web application. The main question is, how can she prove that she owns the NFT?

She could expose her private key to the Ethereum address that holds the NFT to you. It's a valid way but it's an extremely unwise approach. You can steal all assets that that Ethereum address holds if you know the private key.

You can also ask her to do something that originates from her Ethereum address. One example is that you can ask her to transfer a small specific amount of money, such as 0.0000001 ETH, to an Ethereum address of your choosing. If she can do it, it means she owns the Ethereum address that owns the NFT.

But that's not efficient. The user would not be happy. A small amount of money is still money. It's also not scalable. Every time a user needs to prove an NFT ownership, they would need to spend money on the blockchain. Don't forget the transaction fee, which is not really cheap.

You want something that's free and easy to do. The answer is to create a **signature**. With a private key, the owner of an Ethereum address can sign a message. The result of signing a message is a signature. Then, another person can receive the message and the signature, and then verify whether the message is really signed by the owner of the Ethereum address.

Say that a user buys an NFT from your NFT collection that is named "Movies Streaming NFT." The NFT token has an ID of 3. You could send the user a message, `"I own NFT token id 3"`, and ask the user to sign the message. The user would sign the message and give you the signature. You would verify the message with the signature. If it's valid, it means the user owns the Ethereum address that holds the NFT. To verify the ownership, you can check it on the blockchain. The signature only proves the ownership of an Ethereum address outside the blockchain.

To put it in a nutshell, a signature proves the ownership of an Ethereum address outside the blockchain. But you can check the ownership of ERC-20 tokens or NFT tokens on the blockchain. That way, a user outside the blockchain or in a web application can prove their ownership of ERC-20 tokens or NFT tokens using a signature.

Now, let's see how to create a signature and verify a message with the signature in Python code.

Create a Python virtual environment and install the `Ape` library:

```
$ python3.10 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install eth-ape'[recommended-plugins] '== 0.7.23
```

If you don't have an `Ape` account, create one. Otherwise, you can reuse your old account. To create a new `Ape` account, run the following command:

```
$ ape accounts generate chapter_17
Enhance the security of your account by adding additional random
input:
Show mnemonic? [Y/n]: Y
Create Passphrase to encrypt account:
Repeat for confirmation:
INFO: Newly generated mnemonic is: assist eyebrow gold negative
hamster label answer group faint patrol celery metal
SUCCESS: A new account '0x50b177C373Fa60831AD6231B9b885666384D5BB6'
with HDPath m/44'/60'/0'/0/0 has been added with the id 'chapter_17'
```

The `chapter_17` account has been created. Now, let's create a Python file named `sign_message.py` and add the following code:

```
from ape import accounts
from eth_account.messages import encode_defunct
from ape.types.signatures import recover_signer
import os

username = os.environ["ACCOUNT_USERNAME"]
account = accounts.load(username)
password = os.environ["MY_PASSWORD"]
account.set_autosign(True, passphrase=password)
```

```
message = encode_defunct(text="Hello PacktPub!")
signature = account.sign_message(message)
print(f"The address that creates the signature: {account.address}")
print(f"Signature: {signature}")

recovered_signer = recover_signer(message, signature)
print(f"The address that comes from the signature: {recovered_
signer}")
```

To sign the message, you can use the `sign_message` method of the `Account` object. The message must be encoded with the `encode_defunt` function.

To verify the message, you can use the `recover_signer` function that accepts the message and the signature. The result is an address. You can make sure the address is what you expect.

Before executing the script, you must set the environment variables:

```
(.venv) $ export ACCOUNT_USERNAME=chapter_17
(.venv) $ export MY_PASSWORD=yourpassword
```

Make sure to adjust the values because you could have a different password and a different account name.

Then you can execute the Python script this way:

```
(.venv) $ python sign_message.py
WARNING: Danger! This account will now sign any transaction it's
given.
WARNING: Using cached key for chapter_17
The address that creates the signature: 0x50b177C373Fa60831AD6231B9b8
85666384D5BB6
Signature: <MessageSignature v=28
r=0x2c0a95f017f30c016530d318c7d92f4ea2adbf132f733ebe461ec1e45895e457
s=0x6c0504c2a1be2131513e95c937cfda0766624e67f993d04dcc346a4ab4bff296>
The address that comes from the signature: 0x50b177C373Fa60831AD6231B9
b885666384D5BB6
```

You can see that the address that comes from the signature, `0x50b177C373Fa60831AD6231B9b885666384D5BB6`, is the same as the address that signs the message, `0x50b177C373Fa60831AD6231B9b885666384D5BB6`.

This means you can know that `0x50b177C373Fa60831AD6231B9b885666384D5BB6` approves the message `"Hello Packtpub!"`.

This way, you can create a scheme that allows users who have Ethereum addresses to log in to a web application. There is no need to create a username and password. It's like logging in to a web application using a Google account.

You can create a message, `"I want to log in into the XYZ movie streaming web application"`, and ask everyone who wants to log in to sign the message and send the signature to the web application.

Although you use any message in a login process, there is a standard message to be signed if users want to log in to a website.

ERC-4361

The **ERC-4361 standard** describes the implementation of signing in with Ethereum. The full specification can be read at `https://eips.ethereum.org/EIPS/eip-4361`.

The message template looks like this:

```
${scheme}:// ${domain} wants you to sign in with your Ethereum
account:
${address}

${statement}

URI: ${uri}
Version: ${version}
Chain ID: ${chain-id}
Nonce: ${nonce}
Issued At: ${issued-at}
Expiration Time: ${expiration-time}
Not Before: ${not-before}
Request ID: ${request-id}
Resources:
- ${resources[0]}
- ${resources[1]}
...
- ${resources[n]}
```

You can replace `${...}` with values that are suitable for your purpose. For example, `${address}` can be replaced with the Ethereum address that wants to sign in to the website. `${domain}` can be replaced with the domain of the web application. `${chain-id}` can be replaced with the chain of the blockchain. If the blockchain is the Ethereum mainnet, then the chain ID is 1. If it's Optimism, then the chain ID is 10.

The message standard has the **Augmented Backus–Naur Form** (**ABNF**) message format, which looks like this:

```
sign-in-with-ethereum =
    [ scheme "://" ] domain %s" wants you to sign in with your
```

```
Ethereum account:" LF
    address LF
    LF
    [ statement LF ]
    LF
    %s"URI: " uri LF
    %s"Version: " version LF
    %s"Chain ID: " chain-id LF
    %s"Nonce: " nonce LF
    %s"Issued At: " issued-at
    [ LF %s"Expiration Time: " expiration-time ]
    [ LF %s"Not Before: " not-before ]
    [ LF %s"Request ID: " request-id ]
    [ LF %s"Resources:"
    resources ]
...
```

To see the rest of the message standard, you can go to the standard ERC-4361 URL, `https://eips.ethereum.org/EIPS/eip-4361`. This format is useful if you want to parse the message and make sure it follows the grammar of the message.

ABNF is a notation that defines formal language constructs, such as the syntax of programming languages, and in this case, a message to be signed for logging in with Ethereum.

To verify the message and its signature, you can use the SIWE library. First, install the library:

```
(.venv) $ pip install siwe
```

Create a Python script named `sign_in_ethereum.py` and then add the following code:

```
from siwe import SiweMessage
import siwe


eip_4361_string = """
service.org wants you to sign in with your Ethereum account:
0x8e6E9F42fB6d2052cf452A50B3550e1F7A04FaD0

I accept the ServiceOrg Terms of Service: https://service.org/tos

URI: https://service.org/login
Version: 1
Chain ID: 1
Nonce: 32891756
Issued At: 2021-09-30T16:25:24Z
```

```
Resources:
- ipfs://bafybeiemxf5abjwjbikoz4mc3a3dla6ual3jsgpdr4cjr3oz3evfyavhwq/
- https://example.com/my-web2-claim.json
"""
message = SiweMessage.from_message(message=eip_4361_string,
abnf=False)
signature = "0xedb8936240893b956378822f3663d8a9c29f59ac904de2061
08768e5e054b4441cc673af036cef8bb90c0186017fa8db8f1551be97c176374
c0a5bd2cacd27d91c"

try:
    message.verify(signature=signature)
    print("Signature is valid")
except siwe.VerificationError:
    print("Signature is invalid")
```

The message doesn't have `${...}` strings anymore. `${address}` has been replaced with `0x8e6E9F42fB6d2052cf452A50B3550e1F7A04FaD0`. Also, `${domain}` has been replaced with `service.org`. So, in this example, the `0x8e6E9F42fB6d2052cf452A50B3550e1F7A04FaD0` address wants to log in to the service.org website.

You create a SIWE message object from this message with the `from_message` method. The second argument, `abnf`, is given a false value, meaning you don't want to be strict with the ABNF message format.

To verify the message, you can use the `verify` method of the SIWE message object with the signature sent as the argument. If it's valid, then the execution of the method will not throw an exception.

Execute the following Python script:

```
(.venv) $ python sign_in_ethereum.py
Signature is valid
```

But how do you create the signature in the script? You need another application to produce the signature. That application is called **MetaMask**, a wallet browser extension.

Installing a wallet browser extension

To sign a message, of course, you can use a Python script and execute it on the command line. But that's not convenient. There is a more practical way. You can use a wallet browser extension, MetaMask. You can download it from this UR:, `https://metamask.io`.

While MetaMask is not the only wallet browser extension available, it is currently the most widely adopted option among users.

You can install MetaMask in the browser of your choice, such as Mozilla Firefox or Google Chrome. After a successful installation, you should see the following screen:

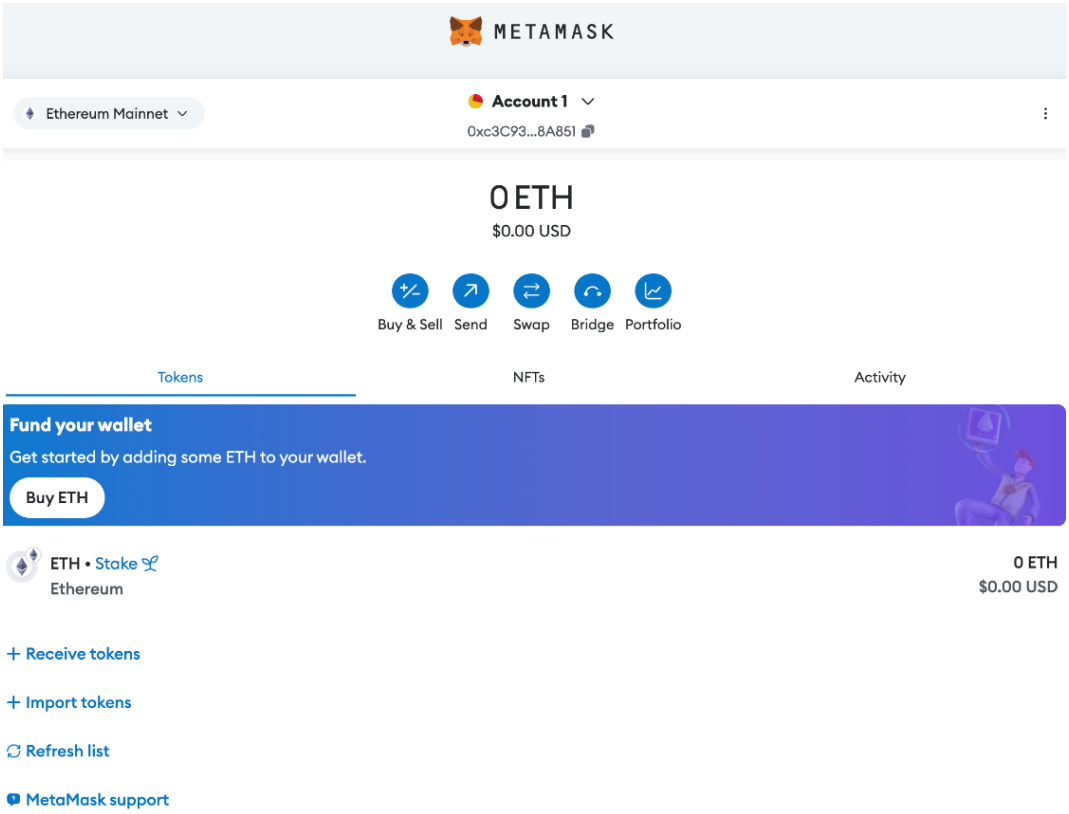


Figure 17.1: The MetaMask application page

The address can be found below the **Account 1** label. You can buy and transfer ETH with this application and can swap ERC-20 tokens, but MetaMask can also be used on a web page.

You'll sign a signature on a web page with MetaMask. But before that, let's deploy an NFT smart contract. You'll use NFT tokens in a token-gated application that you're going to build.

Deploying a token smart contract

If you have read through all the chapters so far, this section will appear to cover what you are already familiar with. However, I have still chosen to include this section for quick reference and to reinforce your learning.

First, run the Ethereum blockchain in development mode:

```
$ geth --dev --http --http.api eth,web3,net
```

Find the location of the IPC file from the output of the command. If you run Geth on Linux, usually the location of the IPC file is `/tmp/geth.ipc`.

Find the address of your `Ape` account:

```
$ cat ~/.ape/accounts/chapter_17.json
{"address": "50b177C373Fa60831AD6231B9b885666384D
5BB6", "crypto": {"cipher": "aes-128-ctr", "cipherparams":
{"iv": "184bd106870064800d1f225f890c402b"}, "ciphertext":
"33f7b733e59f5ff9cdf638eefb35f153154b2a21234ac14c6b80631b7e276a96",
"kdf": "scrypt", "kdfparams": {"dklen": 32, "n": 262144, "r":
8, "p": 1, "salt": "fbd20abc6b36bde289063691efdeab3f"}, "mac":
"bd7d36d3f8512926de90343d86164180f721c76bce9e40bd3305a8e7bd2126a5"},
"id": "6fb78ab0-ebe4-49bc-8900-9f88a9745227", "version": 3}
```

Then, from another terminal, connect to the Ethereum blockchain with Geth:

```
$ geth attach /tmp/geth.ipc
Welcome to the Geth JavaScript console!
...
>
```

In the Geth console, transfer some ETH to the Ethereum address that you get from the `Ape` account:

```
> var recipient = "0x50b177C373Fa60831AD6231B9b885666384D5BB6"
undefined
> eth.sendTransaction({from: eth.accounts[0], to: recipient, value:
web3.toWei(10, "ether")})

"0xd30e324c001e19552c5fb6a60566863871af63b351ba487c27e347243ced7f90"
```

Make sure the recipient variable reflects your `Ape` account's address.

Then create an Ape project:

```
(.venv) $ mkdir token-gated-smart-contract
(.venv) $ cd token-gated-smart-contract
(.venv) $ ape init
Please enter project name: TokenGated
SUCCESS: TokenGated is written in ape-config.yaml
```

You can reuse the `HelloNFT` smart contract that you wrote in *Chapter 13*. For convenience, you can download the smart contract from this URL: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_17/token-gated-smart-contract/contracts/HelloNFT.vy`.

Put the file inside the contracts folder then compile it:

```
(.venv) $ ls contracts
HelloNFT.vy
(.venv) $ ape compile
```

Then, create a deployment script named `deploy.py` inside the `scripts` folder and add the following code:

```
from ape import accounts, project
import os


def main():
    password = os.environ["MY_PASSWORD"]
    username = os.environ["ACCOUNT_USERNAME"]
    deployer = accounts.load(username)
    deployer.set_autosign(True, passphrase=password)
    contract = project.HelloNFT.deploy(sender=deployer)
    print(f"Deployed to {contract.address}")
```

Before executing the script, set the environment variables referring to your Ape account:

```
$ export ACCOUNT_USERNAME=chapter_17
$ export MY_PASSWORD=yourpassword
```

Then, execute the script:

```
(.venv) $ ape run deploy --network ethereum:local:geth
INFO: Connecting to existing Geth node at http://localhost:8545/
[hidden].
WARNING: Danger! This account will now sign any transaction it's
given.
WARNING: Using cached key for chapter_17
INFO: Confirmed
0x6308614f14e6f4309e23f2f68265c26b212ff40f2bd8ae62fec9190e9ac41990
(total fees paid = 580321449339168)
SUCCESS: Contract 'HelloNFT' deployed to:
0x0FC146663896e38092b97593982414cFD9d7dE83
Deployed to 0x0FC146663896e38092b97593982414cFD9d7dE83
```

Make sure you write down the deployed smart contract address. In this example, the deployed smart contract address is `0x0FC146663896e38092b97593982414cFD9d7dE83`. But yours will be different.

Then, create a script named `send_nft.py` inside the `scripts` folder to give an NFT token to a recipient. You want a user who wants to log in to a token-gated web application to have an NFT token that will allow them to access a premium feature. Add the following code to the file:

```
from ape import accounts, project
import os


def main():
    address = os.environ["NFT_ADDRESS"]
    destination = os.environ["NFT_RECIPIENT"]
    password = os.environ["MY_PASSWORD"]
    username = os.environ["ACCOUNT_USERNAME"]
    deployer = accounts.load(username)
    deployer.set_autosign(True, passphrase=password)
    contract = project.HelloNFT.at(address)
    contract.mint(destination, 1, sender=deployer)
```

Look at the last line. You can create an NFT with an ID of 1 and give it to the destination address. The destination address should be the address you get from MetaMask because you will log in to the token-gated web application using MetaMask.

Set up the environment variables:

```
$ export NFT_ADDRESS=<the deployed smart contract address>
$ export NFT_RECIPIENT=<the address in MetaMask>
$ export MY_PASSWORD=<password of the Ape account>
$ export ACCOUNT_USERNAME=chapter_17
```

Replace `chapter_17` with your Ape account username if it's different.

Don't execute this script yet. You want to test what happens if the user doesn't have the NFT token.

Let's build a token-gated web application.

Writing a backend application

To build a token-gated web application, let's use the FastAPI web framework:

```
(.venv) $ cd ../token-gated-smart-contract
(.venv) $ mkdir token-gated-backend
(.venv) $ cd token-gated-backend
(.venv) $ pip install fastapi pyjwt
```

The full code of the token-gated web application can be found here: `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_17/token-gated-backend/token_gated_app.py`.

Create a file named `token_gated_app.py` and add the following code:

```
from datetime import datetime, timedelta, timezone
from string import Template
import string
import json
import secrets
import os
from typing import Union

from ape import Contract
from ape import networks
from ethpm_types import ContractType

from fastapi import Depends, FastAPI, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from fastapi.middleware.cors import CORSMiddleware
import jwt
from pydantic import BaseModel
from typing_extensions import Annotated
from siwe import SiweMessage
import siwe
```

You add a lot of import lines. There are functions and objects from FastAPI, SIWE, Ape, and standard libraries.

Then, add these variables:

```
# to get a string like this run:
# openssl rand -hex 32
SECRET_KEY =
"09d25e094faa6ca2556c818166b7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30
```

These variables are needed to generate a JWT in the backend web application. After a user logs in to the web application, you need to generate a token for the user.

Add the message template:

```
EIP_4361_STRING = Template("""
packtpub.com wants you to sign in with your Ethereum account:
```

```
$address

I accept the PacktPub Terms of Service: https://www.packtpub.com/en-
us/help/terms-and-conditions

URI: http://127.0.0.1:8000/token
Version: 1
Chain ID: 1
Nonce: $nonce
Issued At: $nonce_time
Resources:
- https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-
Developers--2nd-Edition
""")
```

You can substitute the variables (strings with the `$` sign) with values when a user wants to sign in with Ethereum.

Add these lines:

```
nonces_data = {}

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
```

The `nonces_data` variable will hold the nonces data. The `oauth2_scheme` variable will be needed when you want to get a user from the backend token.

Add the models that will be used for the backend API:

```
class Token(BaseModel):
    access_token: str
    token_type: str

class Nonce(BaseModel):
    nonce: str
    nonce_time: str

class Crypto(BaseModel):
    address: str
    signature: str

class User(BaseModel):
    address: str
```

These classes will represent arguments and results from API endpoints.

Let's add a function to create a backend token:

```
def create_access_token(data: dict, expires_delta: Union[timedelta,
None] = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.now(timezone.utc) + expires_delta
    else:
        expire = datetime.now(timezone.utc) + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY,
algorithm=ALGORITHM)
    return encoded_jwt
```

You're using a JWT to generate a token for users. To create a JWT, you can use the `jwt.encode` method that accepts data, the secret key, and the algorithm. The data has the user information and the expiry time.

Let's add a function to generate a nonce:

```
def generate_nonce(length=16):
    chars = string.ascii_letters + string.digits
    return ''.join(secrets.choice(chars) for _ in range(length))
```

In this function, you generate 16 random characters. Later, you'll insert a nonce into a message template to avoid a replay attack. Suppose you want a user to sign this message, `"I want to log in"`. You get the signature and verify it. Everything goes well. But if the signature is leaked to other people, other people can use this signature to log in as the user. So, you embed a nonce into the message so the message would be something like this, `"I want to log in, nonce: 343"`. Next week, if a user wants to log in again, the user will get a different message: `"I want to log in, nonce: 1001"`. That way, the signature cannot be used again in the future.

Let's add the code to start the `FastAPI` web application:

```
app = FastAPI()

origins = [
    "http://localhost:5173",
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"]
)
```

You create a `FastAPI` object using the `FastAPI` function. Later, a frontend application will access this backend application from another server, `localhost:5173`, so you have to allow it explicitly using CORS. You'll build the front-end application after this section.

Let's create the first API endpoint, which is to generate a nonce:

```
@app.get("/nonce/{address}")
async def nonce(address: str) -> Nonce:
    nonce = generate_nonce()
    nonce_time = datetime.now().strftime("%Y-%m-%dT%H:%M:%SZ")
    nonces_data[address] = {"nonce": nonce, "nonce_time": nonce_time}
    return nonces_data[address]
```

The decorator tells us that to access this endpoint, you have to access `/nonce/{address}`. The `{address}` part is the parameter. Inside the function, you'll generate the nonce with the `generate_nonce` function. Then you insert the nonce data into the `nonces_data` variable.

In production, you don't want to use this approach because anyone could generate a nonce for an address that they don't own. You want to generate the nonce implicitly and then send it to the frontend application using session. But for this tutorial, this will suffice.

Let's create the login function:

```
@app.post("/token")
async def login_for_access_token(
  crypto: Crypto,
) -> Token:
    address = crypto.address
    nonce = nonces_data[address]["nonce"]
    nonce_time = nonces_data[address]["nonce_time"]
    signature = crypto.signature
    eip_string = EIP_4361_STRING.substitute(address=address,
nonce=nonce, nonce_time=nonce_time)
    message = SiweMessage.from_message(message=eip_string, abnf=False)
```

The URL of this API endpoint is `/token`. The parameter that the user needs to send to this endpoint is the `Crypto` object that you defined earlier in the code. The result of this API endpoint is a `Token` object. You defined the `Token` model earlier. You get the address and the signature from the `Crypto` object. You get the nonces and the nonce time from the `nonces_data` variable. You substitute the variables inside the message template with these values to get the full message. Then you create a SIWE message from this full message.

Add this code to verify the message:

```
    try:
        message.verify(signature=signature)
        access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_
MINUTES)
        access_token = create_access_token(
            data={"sub": address}, expires_delta=access_token_expires
        )
        return Token(access_token=access_token, token_type="bearer")
    except siwe.VerificationError:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect signature",
            headers={"WWW-Authenticate": "Bearer"},
        )
```

You verify the SIWE message with the signature. If it's valid, then you create a JWT with the `create_access_token` function. You embed the address into the JWT. Then, you return the `Token` object, consisting of the access token. In the frontend, this will be JSON data. If the message verification fails, you raise an exception.

Let's add a function that uses the JWT:

```
async def get_current_user(token: Annotated[str, Depends(oauth2_
scheme)]):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY,
algorithms=[ALGORITHM])
        address: str = payload.get("sub")
        if address is None:
            raise credentials_exception
    except jwt.exceptions.InvalidSignatureError:
        raise credentials_exception
    exists = address in nonces_data
    if not exists:
        raise credentials_exception
    return User(address=address)
```

In the `get_current_user` function, you extract the user from the JWT with the `jwt.decode` method. You need to use the same secret key and algorithm when you encode the data into the token. In the end, you create a `User` object that consists of the address. So, there is an Ethereum address information in the JWT.

Add a function that gets the current user using the `get_currenct user` function:

```
async def get_current_active_user(
    current_user: Annotated[User, Depends(get_current_user)],
):
    return current_user
```

After the get_current_active_user function is ready, then you can create the function that serves the `/me` path so a user can get the information about them:

```
@app.get("/me", response_model=User)
async def read_users_me(
    current_user: Annotated[User, Depends(get_current_active_user)],
):
    return current_user
```

You create the `/me` API endpoint that will return information about the current active user.

The last function that you need to add is the function to read the content, whether basic content or premium content:

```
BLOCKCHAIN_NETWORK = "local"
BLOCKCHAIN_PROVIDER = "geth"

@app.get("/content")
async def read_content(
    current_user: Annotated[User, Depends(get_current_active_user)],
):
    with open('../token-gated-smart-contract/.build/HelloNFT.json') as
f:
        contract = json.load(f)
        abi = contract['abi']

    nft_address = os.environ["NFT_ADDRESS"]
    with networks.ethereum[BLOCKCHAIN_NETWORK].use_
provider(BLOCKCHAIN_PROVIDER):
        ct = ContractType.parse_obj({"abi": abi})
        NFTSmartContract = Contract(nft_address, ct)
        own_nft = NFTSmartContract.balanceOf(current_user.address) > 0
```

```
    if own_nft:
        return {"content": "Premium content"}
    else:
        return {"content": "Basic content"}
```

The API endpoint is `/content` in this function. You get the user information from the `get_current_active_user` function. You load the ABI of the NFT smart contract from the Ape project directory. Make sure the token-gated backend project directory is in the same directory as the token-gated smart contract project directory. You create a smart contract object with the deployed NFT smart contract address. Then you check whether the user's address has an NFT token from this NFT smart contract. You use the `balanceOf` method to achieve that purpose inside the `with` block. The `with` block uses the local Ethereum network with Geth provider. If the address has an NFT, then you return the premium content. If not, you return the basic content.

Set an environment variable to tell the web application where to find the NFT smart contract:

```
$ export NFT_ADDRESS=<the deployed NFT smart contract address>
```

To run the token-gated web application, run the following command:

```
(.venv) $ fastapi dev token_gated_app.py
```

The application will run on `http://localhost:8000`.

The backend application is ready. It's time to build the front-end application.

Writing a frontend application

To create a frontend application that interacts with the backend web application that you've just written, let's use the `Wagmi` library. But first, you need to install some additional tools and libraries.

To install Node/js, go to `https://nodejs.org/en/download`. Follow the instructions on the website. To verify that you have a working Node.js instance, run the following command:

```
$ node --version
v20.12.1
```

Then install the **pNpM** package manager. You can get the software from `https://pnpm.io/`.

On Linux or macOS, you can run the following command to install pNpM:

```
$ curl -fsSL https://get.pnpm.io/install.sh | sh -
```

Then create a frontend project using the following command:

```
$ pnpm create wagmi
.../18f8aec0d01-2163                      |  +12 +
.../18f8aec0d01-2163                      | Progress: resolved 12,
reused 0, downloaded 12, added 12, done
✔ Project name: … token-gated-blog
✔ Select a framework: › React
✔ Select a variant: › Vite

Scaffolding project in /Users/arjuna/Code/book/Hands-On-Blockchain-
for-Python-Developers--2nd-Edition/chapter_17/token-gated-blog...

Done. Now run:

  cd token-gated-blog
  pnpm install
  pnpm run dev
```

Here, you gave the project the name `token-gated-blog`, chose the React framework, and chose the Vite variant.

Run the following command:

```
$ cd token-gated-blog
$ pnpm install
$ pnpm run dev
```

The frontend application runs on `http://localhost:5173/`.

The full code of the frontend application can be found at `https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-Developers--2nd-Edition/blob/main/chapter_17/token-gated-blog/src/App.tsx`.

Open `token-gated-blog/src/App.tsx`. Replace the content of the file with `App.tsx` from the GitHub repository shared in the previous paragraph.

To interact with MetaMask, you need to use functions from the `Wagmi` library as in the following code:

```
  const account = useAccount()
  const { connectors, connect, status, error } = useConnect()
  const { disconnect } = useDisconnect()
  const { signMessageAsync } = useSignMessage()
```

To use MetaMask, you'll use the `connect` function. To sign a message, you'll use the `signMessageAsync` function.

The message template needs to be the same as the message template in the backend web application, as seen in the following code:

```
  const message = `
packtpub.com wants you to sign in with your Ethereum account:
$address

I accept the PacktPub Terms of Service: https://www.packtpub.com/en-
us/help/terms-and-conditions

URI: http://127.0.0.1:8000/token
Version: 1
Chain ID: 1
Nonce: $nonce
Issued At: $time
Resources:
- https://github.com/PacktPublishing/Hands-On-Blockchain-for-Python-
Developers--2nd-Edition
`;
```

To get the nonce from the backend web application, here's the relevant code:

```
  const fetchNonce = async () => {
    try {
      const address = account.addresses[0]
      const response = await fetch(`http://localhost:8000/
nonce/${address}`,
                                   {method: 'GET',
                                    headers: {'Content-Type':
'application/json'},
                                   })
      const nonceData = await response.json()
      const nonceValue = nonceData.nonce
      const nonceTimeValue = nonceData.nonce_time

      setNonce(nonceValue)
      setNonceTime(nonceTimeValue)
    } catch (error) {
      console.error('Error fetching nonce:', error)
    }
  };
```

You get the nonce from the GET request to the backend web application and then you set the nonce data into variables using `setNonce`.

To use MetaMask, here's the relevant code:

```
      <div>
        <h2>Connect</h2>
        {connectors.map((connector) => (
          <button
            key={connector.uid}
            onClick={() => connect({ connector })}
            type="button"
          >
            {connector.name}
          </button>
        ))}
        <div>{status}</div>
        <div>{error?.message}</div>
      </div>
```

Upon calling the `connect` function, the **MetaMask** window will show up, asking you to unlock using the password.

To sign in with Ethereum, you need to call `signMessageAsync`, as seen in the following code:

```
        <button type="button"
                onClick={async () => {
                        const address = account.addresses[0]
                        const finalMessage = message.
replace(/\$address/g, address)
                                                    .
replace(/\$nonce/g, nonce)
                                                    .
replace(/\$time/g, nonceTime)
                        const signedMessage = await
signMessageAsync({ message: finalMessage })
                        setSignature(signedMessage)
```

You can feed the address and nonce to the message template to get the full message. Then, you sign the message using the `signMessageAsync` function. This function will trigger the **MetaMask** window. You'll need to confirm the signing-message action with MetaMask.

Once you sign the message with MetaMask, you need to send the address and the signature to the backend application as seen in the following code:

```
                          const tokenRes = await fetch('http://
localhost:8000/token', {
                                                              headers:
{'Content-Type': 'application/json'},
                                                              method: 'POST',
                                                              body: JSON.
stringify({ address, signature: signedMessage })
                                                        });
                          const tokenDict = await tokenRes.json()
                          setToken(tokenDict["access_token"])
                 }}>
            Sign message
          </button>
```

As you can see, you send the signature and the address to the `/token` path. If the signature and address have been verified successfully in the backend application, you'll receive the access token.

To get the content from the backend application, you can use this JWT. Here's the relevant code:

```
           <button
             onClick={async () => {
                          const contentRes = await fetch('http://
localhost:8000/content', {

               headers: {'Content-Type': 'application/json',

           'Authorization': `Bearer ${token}`,},
                                                        method: 'GET'})
                          const contentValue = await contentRes.json()
                          setContent(contentValue["content"])
             }}
             type="button"
           >
             Read Content
           </button>
```

You put the JWT in the `Authorization` header in the request to the `/content` endpoint. The backend application will extract the address from this JWT. Then, you'll get the content.

Let's see the web front end. Go to `http://localhost:5173`. You'll see the following screen:

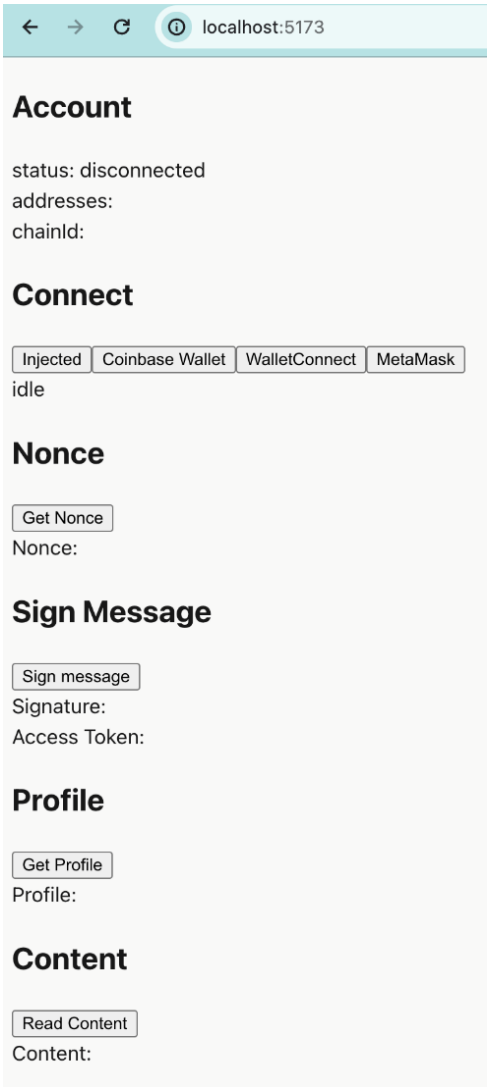


Figure 17.2: The web frontend

Click the **MetaMask** button below the `Connect` label. The **MetaMask** window will show up like this:

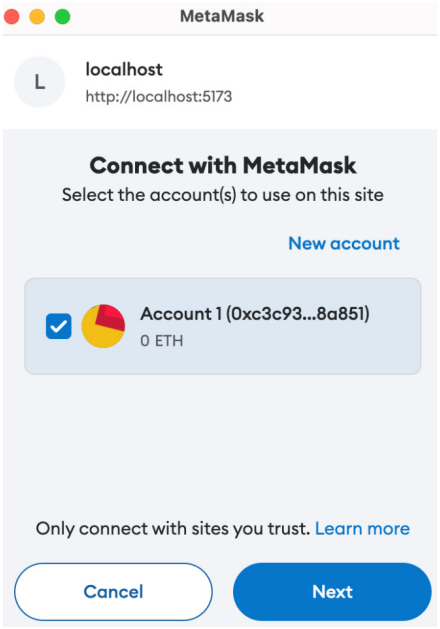


Figure 17.3: The Connect with MetaMask window

Click the **Next** button and you will see the following screen on MetaMask:

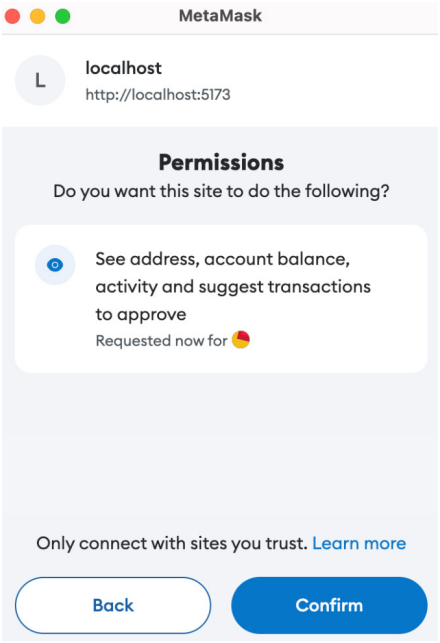


Figure 17.4: The Permissions window

Click on the **Confirm** button, and then you'll see the success message on the web page.

Now, click the **Nonce** button. You'll get the nonce value from the backend server as seen in the following screen:

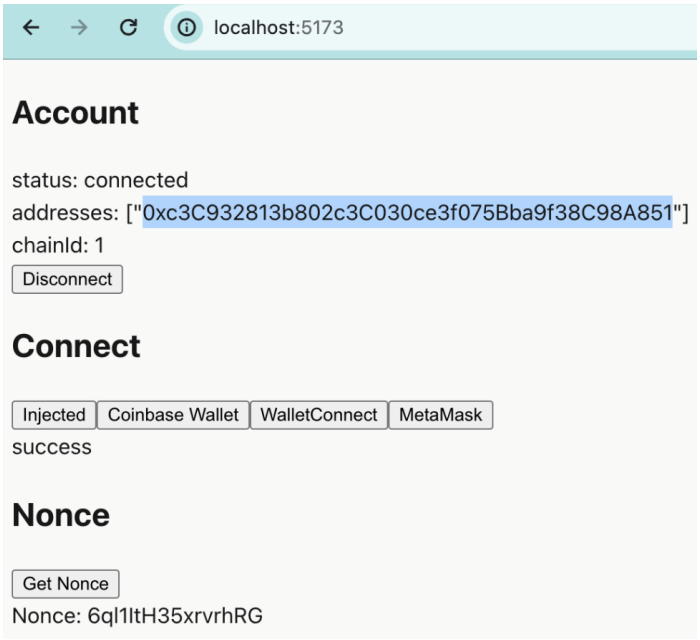


Figure 17.5: The nonce value

Then you can click the **Sign message** button. The **MetaMask** window will appear again like this:

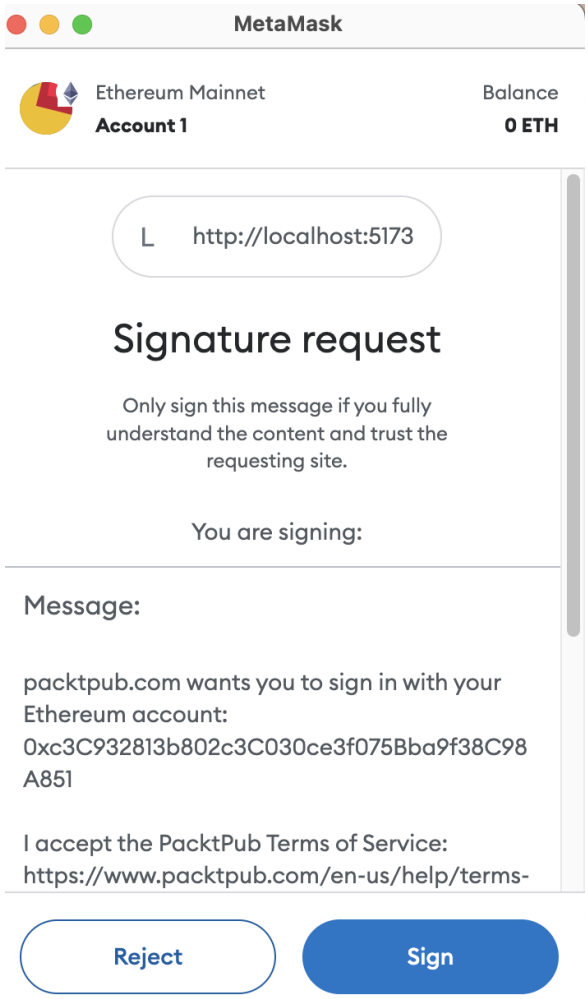


Figure 17.6: The Signature request window

You can see the message that you want to sign. The message contains the address from MetaMask. Click the **Sign** button. Then you'll get the JWT.

Lastly, you can use this JWT to get the content from the token-gated web application. Click the **Read Content** button, and then you'll get the content as seen in the following screenshot:

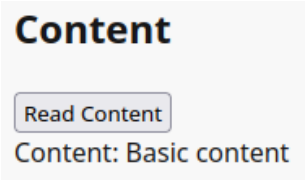


Figure 17.7: The basic content

Remember that the premium content is gated to owners of NFTs. Let's give an NFT to this user:

```
(.venv) $ cd token-gated-smart-contract
(.venv) $ ape run send_nft --network ethereum:local:geth
```

Make sure all environment variables have been set properly before you run the script.

Click the **Read Content** button again. This time, you will able to read the premium content because you have the NFT as seen in the following screenshot:

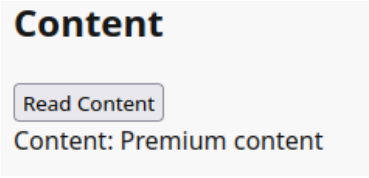


Figure 17.8: The premium content

You've finally built a token-gated web application successfully. You have brought blockchain technology to a web application!

Summary

In this chapter, you learned the concepts of token-gated applications. You saw that the tokens from a blockchain determine the access for an application outside the blockchain. Then you learned how to prove the ownership of tokens with a signature. You installed a wallet browser extension. After that, you deployed an NFT smart contract before writing a token-gated web application. Finally, you wrote the frontend application that signs a message using a browser extension and logs in to the token-gated web application. You gave an NFT to the user in the browser extension so the user could access the gated content. Finally, you mastered the knowledge of blockchain programming!

Throughout this book, you have gained a comprehensive understanding of blockchain programming, from fundamental concepts to practical implementation. By mastering the skills and knowledge presented, you are now equipped to navigate the rapidly evolving landscape of blockchain technology. The concepts you have learned, such as smart contracts, IPFS, ERC-20 tokens, NFTs, decentralized exchanges, NFT marketplaces, and token-gated applications, are at the forefront of innovation in finance, digital identity, and numerous other domains. However, don't stop here. The opportunities are many. You are just swimming at the surface level of blockchain technology. Let these skills and knowledge guide you toward solving real problems with blockchain.

Index

D

E

F

M

N

O

P

R

W

X

Z

packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

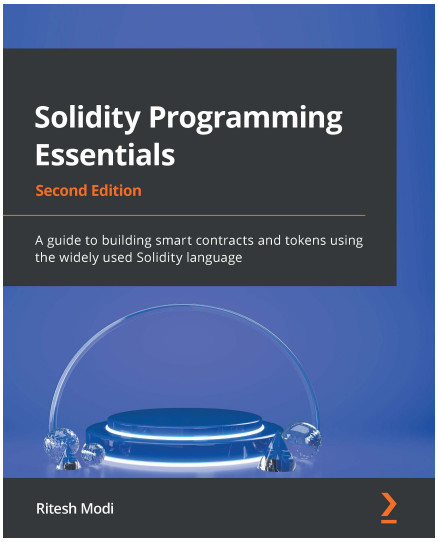
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Solidity Programming Essentials

Ritesh Modi

ISBN: 978-1-80323-118-1

- Write efficient, effective, and secure smart contracts
- Code, compile, and test smart contracts in an object-oriented way
- Implement assembly code in Solidity
- Adopt upgradable and haltable ownership and security design patterns
- Understand exception handling and debugging in Solidity
- Create new ERC20 and NFT tokens from the ground up

Applied Computational Thinking with Python

Sofía De Jesús, Dayrene Martinez

ISBN: 978-1-83763-230-5

- Find out how to use decomposition to solve problems through visual representation
- Employ pattern generalization and abstraction to design solutions
- Build analytical skills to assess algorithmic solutions
- Use computational thinking with Python for statistical analysis
- Understand the input and output needs for designing algorithmic solutions
- Use computational thinking to solve data processing problems
- Identify errors in logical processing to refine your solution design
- Apply computational thinking in domains, such as cryptography, and machine learning

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Hands-On Blockchain for Python Developers - Second Editon* , we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781805121367`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly