CHAPTER 2

The Ethereum Development Environment

This chapter walks you through the setup and installation of tools required to run the Ethereum blockchain. We cover hardware requirements, operating system requirements, and software requirements. After covering the installation of the software, we provide the basic commands required to interact with the Ethereum network.

Getting Set Up

Coders who have set up a development environment for a compiled language in the past will find the setup for Solidity to be a similar process. Setting up Solidity and the associated tools requires some knowledge of the command line and a UNIX-derived operating system. For first-time developers or those with no command-line experience, we recommend going through the Learn Enough Command Line to Be Dangerous online tutorial (`www.learnenough.com/command-line-tutorial`) before tackling Solidity.

K. Iyer and C. Dannen, *Building Games with Ethereum Smart Contracts*,
https://doi.org/10.1007/978-1-4842-3492-1_2

Hardware Choices

The primary hardware requirements for any blockchain development, not just Ethereum, are a reliable Internet connection and large hard drive.

Syncing a copy of the blockchain with a good Internet connection can take up to 8 hours, though this operation has to be completed only once. Just for the one-time sync, it is recommended to find a minimum 5Mbps download connection to connect to for a night. Syncing on a slower connection, while possible, will simply take longer. Broadcasting transactions, communicating with peers, and downloading new block information all require an always-on, but not necessarily high-bandwidth Internet connection. A connection of 1Mbps download and 512kbps upload should be sufficient for day-to-day operation.

The Ethereum blockchain is large and continually expanding. Running a full archive node, as of December 2017, takes 350GB of disk space.[1] Thankfully, we can run a full node with just the latest snapshot of the state tree, which as of December 2017 occupies only 35GB of disk space. Maintaining the state tree snapshot after syncing requires the equivalent of syncing an archive node from the current block forward. Ideally, you would have 400GB available, but 75GB is the bare minimum you would need available to run a full node.

In addition to the hard disk size, your hard disk must be a solid-state drive (SSD). Using a traditional seeking disk drive (HDD) will be too slow. Any computer manufactured since 2010 will have sufficient compute power and RAM, so those should not be an issue.

[1] "Ethereum Database Size", http://bc.daniel.net.nz/

Operating System

All the terminal (command-line) commands in this book are geared toward users on UNIX-derived operating systems. In modern speak, that means if you are running Mac or Linux, you should be fine. Windows users will not find this book difficult to follow, as most of the commands and code are the same across all systems, but should you choose to use Windows, you will be on your own for the installations in the remainder of this section.

Tip To make following along with the book easier, Windows users can install GNU on Windows, a series of UNIX shell utilities ported over to Windows. The installer can be downloaded from `https://github.com/bmatzelle/gow/wiki`.

Linux

All variants of Linux (Ubuntu, Debian, Red Hat, Arch Linux) already have the necessary tools required to run an Ethereum client. We will be spending a lot of time operating in the command-line interface (CLI). All Linux systems have a built-in CLI program with a name like Terminal, Bash, or Shell. Some variants of Linux are CLI-only. Most aren't. In many Linux systems, the shortcut to access the terminal is Ctrl+Alt+T.

In this book, installation instructions for the required CLI programs are included for both the apt and yum package managers. Package managers make it easy to install other programs and dependencies from the command line. Most Linux distributions come with either apt or yum built in. If you are not sure about which one you have, type both commands into your CLI and see which one works. Figure 2-1 shows the output of the built-in apt manager on Ubuntu.

```
kedar@kedar-Latitude-E6430:~$ apt
apt 1.2.15 (amd64)
Usage: apt [options] command

apt is a commandline package manager and provides commands for
searching and managing as well as querying information about packages.
It provides the same functionality as the specialized APT tools,
like apt-get and apt-cache, but enables options more suitable for
interactive use by default.

Most used commands:
  list - list packages based on package names
  search - search in package descriptions
  show - show package details
  install - install packages
  remove - remove packages
  autoremove - Remove automatically all unused packages
  update - update list of available packages
  upgrade - upgrade the system by installing/upgrading packages
  full-upgrade - upgrade the system by removing/installing/upgrading packages
  edit-sources - edit the source information file

See apt(8) for more information about the available commands.
Configuration options and syntax is detailed in apt.conf(5).
Information about how to configure sources can be found in sources.list(5).
Package and version choices can be expressed via apt_preferences(5).
Security details are available in apt-secure(8).
                                        This APT has Super Cow Powers.
kedar@kedar-Latitude-E6430:~$
```

Figure 2-1. *An Ubuntu CLI with apt installed*

If you are a Windows user and would like to try Linux for this book, your first hurdle is getting a Linux distribution installed on your computer. Many detailed tutorials on the Internet indicate how to do so, so we don't cover that here. If you choose to go this route, we recommend using Ubuntu 16.04 LTS with VirtualBox. Ubuntu is the most beginner-friendly version of Linux, and VirtualBox allows you to run a virtual version of Linux without the pain and hassle of partitioning your hard drive and setting up a dual boot.

macOS

Under the hood, macOS and Linux are similar operating systems. Both are descended from UNIX, an operating system developed by Bell Labs in the 1970s. The built-in CLI program in macOS is called *Terminal*, and it has many of the same commands as its Linux counterpart.

Note Mac, or Macintosh, is the name of the computer produced by Apple, and macOS is the operating system that runs on a Mac. Because the two are always sold together, their names are often used interchangeably.

For our purposes, the key difference between the two CLI environments is the lack of a package manager for macOS. Let's fix that by installing Homebrew. Open the Terminal (you should be able to open it from the Spotlight search pop-up, which can be opened with the shortcut Command+spacebar) and copy in the following command and then press Enter to run the installation:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.\
githubusercontent.com/Homebrew/install/master/install)"
```

When the installation is complete, type `brew` into Terminal. You should see a list of available commands.

Programmer's Toolkit

A few basic programming tools are required for any programming project: text editor, compiler/runtime, version control. Let's get these installed before we dive into Ethereum clients.

Text Editor

A *text editor* is a tool for editing plain text. *Plain text* is a format enabling every letter or symbol to be encoded directly into binary. Code and CLIs operate in plain text because it is the simplest compromise between humans who like pretty things and computers that want everything as 0s and 1s. Most word processors do not actually edit plain text. Microsoft Word uses a proprietary format to allow for advanced styling and formatting and because Microsoft likes making it difficult for users to leave its platform.

Any standard text editor will be good enough for Solidity development. For those who haven't used a text editor before, Sublime Text or Atom will be a good start. For the Java-heads and mobile developers used to an integrated development environment (IDE), there is an IDE for Ethereum development called Remix, but it has limited functionality and most developers don't use it.

Version Control: git

Version control is an essential tool used to back up code, efficiently track changes in a codebase, and enable clean collaboration between multiple developers. Git is the most popular version control system (VCS). Originally developed by Linus Torvalds to manage the Linux kernel source code, git is now used by the vast majority of software projects.

Note We will be using git to connect with this book's official GitHub repository at `https://github.com/k26dr/ethereum-games`. The official GitHub repo contains all the project code and links for this book, and will be updated regularly as the Ethereum ecosystem evolves.

Follow Listing 2-1 to install git.

Listing 2-1. Installing git

```
// macOS
brew install git

// Linux
sudo apt-get install git
```

Runtime: JavaScript

The official client library for interacting with an Ethereum node via RPC is web3.js. To use it, we need to install Node.js and NPM. Imagine you dug through the Chrome browser source code, pulled out just the JavaScript engine, and turned it into a command-line program. That's how Ryan Dahl created Node.js, JavaScript's server-side sister. Node.js uses a module system to organize code, similar to Java or Python or Swift. NPM, Node.js Package Manager, was created to streamline this process and make sharing modules via the Web easy. Think of it as apt or yum for Node.js modules. To install, follow Listing 2-2.

Listing 2-2. Installing Node.js and NPM

```
// macOS
brew install node

// Linux w/ apt
// The second line creates a shortcut from the node command
// to the nodejs program for consistency with the macOS
// package name
sudo apt-get install nodejs npm
sudo ln -s /usr/bin/nodejs /usr/bin/node
```

Compiler: Solidity

Solidity is a compiled language that compiles into EVM bytecode similar to Java. The Solidity compiler will be the first NPM package we install. Install it globally with the following:

```
sudo npm install -g solc
```

Ethereum Clients

The *Ethereum client* is the program that implements the Ethereum protocol and interacts with the Ethereum network and blockchain. Here are some of its responsibilities:

- Sync new chains
- Download and verify new blocks
- Connect to peers
- Verify and execute transactions
- Broadcast local transactions to the network
- Provide basic mining ability

There are multiple Ethereum clients, each with its own pros and cons. We will be using two in this book, geth and TestRPC, but cover two more, Eth and Parity, so you can be familiar with them.

Geth

Geth is the official Go implementation of the Ethereum protocol. It is the most up-to-date Ethereum client and serves as the reference client for all Ethereum updates. As the official reference implementation for Ethereum, geth has all the latest security patches and updates. To install geth, follow Listing 2-3.

Listing 2-3. Installing geth

```
# This is a comment
# Any lines starting with '#' will be ignored by the CLI

# For Linux w/ apt
sudo apt-get install software-properties-common
```

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

# For Mac
brew tap ethereum/ethereum
brew install ethereum
```

TestRPC

TestRPC is a lightweight Ethereum client that specializes in running private chains for development. We will use it to create private networks that are sandboxed from the mainnet. It is built into the Truffle framework, and we cover it along with Truffle later in this chapter.

Eth

Eth is the official C++ implementation of the Ethereum protocol. It is used in applications such as mining that require high performance. It used to support the mining algorithm itself, but that portion of the codebase has since been spun off into its own project called *Ethminer*.

Parity

Parity is a third-party Ethereum client that aims to provide a user-friendly alternative to the geth client and Mist browser. Its development is led by Gavin Wood, an Ethereum cofounder and a prominent member of the community. Parity is targeted at Ethereum users rather than developers and tends to lag geth in having the latest features.

Deployment

Ethereum has two types of addresses: wallet addresses and contract addresses. They look and act the same, but one belongs to a user, and one belongs to a contract. Only the owner of the private key can send the ether belonging to a wallet address. A contract address can have a balance, just like a wallet address. Only the contract code can send the ether belonging to the contract.

Creating a contract is simple in theory; send the contract bytecode to the null address (`0x`). In practice, though, going from a Solidity contract to EVM bytecode with a hand-rolled process is a messy affair, so we're going to pull in one more dependency to simplify the process.

Introducing Truffle

Truffle is a development framework for Solidity and the EVM. Truffle will take care of compiling, deploying, and testing our contracts and allow us to focus on writing the game contracts. To install Truffle globally, use this command:

```
sudo npm install -g truffle
```

Let's get a feel for Truffle by running some basic commands. We will go more into the theory of how all this works later. For now, we're going to deploy our first contract to a private chain. Run the commands in Listing 2-4 in the order provided. The `truffle develop` command will open a Truffle development console running TestRPC. The `migrate` command should be run in that console.

Tip Windows users should use `truffle.cmd` instead of `truffle` for Truffle commands. As an example, `truffle.cmd develop` would open the Truffle dev console.

Listing 2-4. Deploying a sample dapp with Truffle

```
mkdir truffle-test
cd truffle-test
truffle init
truffle develop

# Run this command in the Truffle dev console
migrate

# Exit the dev console
.exit
```

`truffle init` scaffolds a series of folders and sample files, one of which is the `contracts` folder. You should see a Solidity contract file in there: `Migrations.sol`. Take a quick browse through the code in the file. That is the code we just deployed, and reading through it will give you a feel for how Solidity contracts are structured.

Migrating is the Truffle equivalent of deploying. A migration in Truffle is essentially a deployment script. One of the directories scaffolded by Truffle is the `migrations/` folder. There should be a sample migration file in there as well. Take a look at it to see what a simple migration looks like.

Congratulations! You've set up a development chain for yourself and deployed your first Solidity contract.

Basic Geth Commands

Geth is an in-depth program that handles a large deal of functionality. Run `geth help` to see a full list of commands available in geth. It is quite comprehensive. We're going to focus on a small subset of essential commands in this section.

The first command we're going to try out is no command. Run `geth` with no options or commands. You should see something similar to Figure 2-2. Geth is starting up for the first time, connecting to peers, and beginning the sync process. Use Ctrl+C to exit geth.

```
kedar@kedar-Latitude-E6430:~$ geth
INFO [09-28|14:16:00] Starting peer-to-peer node               instance=Geth/v1.6.7-stable-ab5646c5/linux-amd64/go1.8.1
INFO [09-28|14:16:00] Allocated cache and file handles         database=/home/kedar/.ethereum/geth/chaindata cache=128 ha
INFO [09-28|14:16:00] Initialised chain configuration          config="{ChainID: 1 Homestead: 1150000 DAO: 1920000 DAOSupp
000 Metropolis: 9223372036854775807 Engine: ethash}"
INFO [09-28|14:16:00] Disk storage enabled for ethash caches   dir=/home/kedar/.ethereum/geth/ethash count=3
INFO [09-28|14:16:00] Disk storage enabled for ethash DAGs     dir=/home/kedar/.ethash                count=2
INFO [09-28|14:16:00] Initialising Ethereum protocol           versions="[63 62]" network=1
INFO [09-28|14:16:00] Loaded most recent local header          number=4256707 hash=e61711…410d6d td=891172962707076110768
INFO [09-28|14:16:00] Loaded most recent local full block      number=4256707 hash=e61711…410d6d td=891172962707076110768
INFO [09-28|14:16:00] Loaded most recent local fast block      number=4256707 hash=e61711…410d6d td=891172962707076110768
WARN [09-28|14:16:00] Blockchain not empty, fast sync disabled
INFO [09-28|14:16:00] Starting P2P networking
INFO [09-28|14:16:02] UDP listener up                          self=enode://4d6897fab3e0de4a67cf8e1126a1245a2cf80331003c62
825d7775276a26a0b3b62f80844a@[::]:30303
INFO [09-28|14:16:02] RLPx listener up                         self=enode://4d6897fab3e0de4a67cf8e1126a1245a2cf80331003c62
825d7775276a26a0b3b62f80844a@[::]:30303
INFO [09-28|14:16:02] IPC endpoint opened: /home/kedar/.ethereum/geth.ipc
INFO [09-28|14:16:22] Block synchronisation started
```

Figure 2-2. *Geth on startup*

To interact with geth, we need to open geth in console mode. Let's do so with the command `geth console`. You should see something like Figure 2-3 pop up.

```
kedar@kedar-Latitude-E6430:~$ geth --verbosity 0 console
Welcome to the Geth JavaScript console!

instance: Geth/v1.6.7-stable-ab5646c5/linux-amd64/go1.8.1
coinbase: 0xf2e6b44e0ffd524bd36cae1a58d9f6ee2edffb1e
at block: 4256707 (Sat, 09 Sep 2017 18:27:32 EDT)
 datadir: /home/kedar/.ethereum
 modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0

> web3.eth.accounts
```

Figure 2-3. *Geth console*

The geth console exposes a series of modules that allow us to interact with geth. This includes functionality for creating wallets, sending ether, creating contracts, interacting with contracts, and more. As an example, to view a list of our wallets, we could input `eth.accounts` into the console.

We don't have any wallets generated at the moment, so we receive back an empty array. We will be generating wallets and obtaining ether in Project 3-1, and we will revisit the geth console and its many commands at that time. Type `exit` in the console to quit the program.

Many users find the log messages flowing across the geth console to be distracting. To silence the log messages, run the console in silent mode with `geth --verbosity 0 console`.

In addition to the mainnet, geth can be used to access testnets, run private nets, and interact with any other network that observes the Ethereum protocol. We will regularly be connecting to the Rinkeby testnet in this book to test and deploy contracts without having to use any of our precious ether. To connect to the Rinkeby testnet, run `geth --rinkeby`. This will connect to Rinkeby peers and begin the sync process for the Rinkeby network.

Account and wallet management is one of the core features of geth, especially for nondevelopers. To access the account management interface, run `geth account`. This will pull up a help page and list of subcommands that can be used for account management. Let's test one of the commands right now by running `geth account list`. Just as in the console section, you will receive an empty response. The command `geth account new` can be used to create a new account, but we will hold off on doing so until later in the chapter.

To communicate with dapps and external clients, geth can run a JSON-RPC server. To run geth in RPC mode, use `geth --rpc`. For security reasons, RPC mode by default disables access to local private keys. We will be needing RPC access to our private keys to sign and send transactions, so we will run the RPC server with `geth --rpc --rpcapi web3,eth,net,personal`. The personal module enables access to account services.

Caution Enabling the personal RPC API exposes your geth wallets to the Internet. The only thing preventing others from stealing your ether will be your wallet password. Make sure it is strong. We will be repeating this warning multiple times throughout the book.

Sometimes we will want to run two networks at the same time. Later in the chapter, we will be doing this to sync both the mainnet and Rinkeby testnet at the same time. By default, geth connects to port 30303 for network actions and 8545 for the RPC server. Only one program can be listening on a port at a time, so attempting to run two instances of geth at the same time will fail by default. To have one of the instances listen on a different network port (say, 31303), run `geth --port 31303`. To have one of the RPC servers run on a different port (say, 9545), run `geth --rpc --rpcport 9545`.

Docs and Resources

The geth docs can be found on GitHub at https://github.com/ethereum/go-ethereum/wiki/geth The page has links to both the geth console API and geth command reference.

Table 2-1 is a reference for useful geth commands. Some are covered in this chapter, and others are not covered until later chapters but are included here for completeness.

Table 2-1. *Useful Geth Commands*

Description	Command
Default geth mode, used for basic operation	`geth`
Interactive console (silent mode)	`geth console --verbosity 0`
Command reference	`geth help`
Rinkeby testnet	`geth --rinkeby`
Account management	`geth account`
Create account	`geth account new`
Sync mainnet	`geth --fast --cache=1024`
Sync Rinkeby	`geth --rinkeby --fast --cache=1024`
RPC mode	`geth --rpc`
RPC mode with local wallet access	`geth --rpc --rpcapi web3,eth,net,personal`
Listen on custom network port	`geth --port <port>`
Listen on custom RPC port	`geth --rpc --rpcport <port>`

Connecting to the Blockchain

To execute contract deployments and network transactions, you have to sync a full node for each network you wish to use. We will be syncing two networks for this book: the Ethereum main network (mainnet) and the Rinkeby test network (testnet). A *test network* is a network that runs the Ethereum protocol, but whose token has no value. It's useful for testing code, deployments, and transactions without paying gas fees, which can be prohibitively expensive for repetitive testing.

Every public Ethereum network has a unique network ID. The network ID of the Ethereum mainnet is 1. The network ID of the Rinkeby testnet is 4. The network ID of our private chains will be large, random numbers whose only job is to be unique enough to avoid syncing with other networks.

Network Synchronization

Geth offers three modes for network synchronization: light, full, and archive.

A *light node* syncs block headers, but does not process transactions or maintain a state tree. Light clients are useful for users who wish only to maintain wallets and send/receive ether. For developers, a light client will be insufficient; we will require a full node.

A *full node* maintains a local snapshot of the blockchain state tree, downloads full blocks, executes block transactions on its local copy of the blockchain, and participates in the consensus process. Full nodes are the backbone of the Ethereum network. For those of you familiar with torrents, think of the full vs. light client dynamic as analogous to seeds vs. leeches. Full nodes seed network information to peers, whereas light nodes leech information from the network without seeding anything back. Syncing a full node is a slow process that takes about 8 hours and consumes about 30GB of disk space.

An *archive node*, sometimes referred to as a *full archive node*, maintains not only a current snapshot of the state tree, but also a copy of every state transition that has occurred on the chain since the genesis block. A full archive node is the granddaddy of Ethereum nodes, and as of December 2017, consumes 350GB of space while growing at a rate of 30GB per month. If syncing a full node is a slow process, syncing an archive node is damn near impossible. Estimates on my laptop with a standard SSD and 10Mbps Internet connection placed the sync time at 45 days. For those

wishing to run an archive node, your best bet is to use geth's import/export functionality to make a copy of the database from an existing archive node.

We will be syncing full nodes for the mainnet and Rinkeby testnet.

Mainnet

To sync a full node on the mainnet, run the following:

```
geth --fast --cache=1024
```

A fast sync will sync a full node without archives. This process takes about 8 hours on a 10Mbps or faster Internet connection with an SSD drive. Using an HDD takes two to three times longer. The same goes for connections below 3Mpbs. Leave the sync running overnight if you can, and you should be ready in the morning. To save time, you can sync both the mainnet and the testnet at the same time. We explain how to do so in the next section.

Testnet

For the testnet, we will be syncing to the Rinkeby testnet. Past testnets for Ethereum include Olympic, Morden, Ropsten, and Kovan. The Kovan testnet is still active but has been mostly supplanted by the Rinkeby testnet. The other testnets have all been abandoned. Maintaining a testnet turns out to be quite a difficult task, and they get successfully attacked quite regularly. More on this can be found in the "Testnet Attacks and Issues" section in Chapter 6.

We assume that most of you will be syncing the testnet at the same time as the mainnet, so we will run the testnet sync on a different port:

```
geth --rinkeby --port 31303
```

Leave both networks to sync overnight, and resume the exercises in this book when the syncs are complete.

Faucets

Mainnet Ether can be purchased on an exchange with bitcoin or fiat currency, but no exchange will list testnet ether because it has no value. To solve this problem, most testnets use faucets. *Faucets* are sites that send you free crypto. They originated in the early bitcoin days as a quick way for users to obtain small amounts of bitcoin to get a feel for the technology, but faded away after the coin gained serious value. Nowadays they are used seriously only for testnets.

Summary

Running an Ethereum node requires a large solid-state hard drive and a good Internet connection. 75GB of SSD disk space and a 5Mpbs connection are an ideal minimum.

The best operating system for developing Ethereum smart contracts is Linux, with macOS a close second. If you must use Windows, make sure to download GNU on Windows and use `truffle.cmd` instead of `truffle` for your Truffle commands.

An Ethereum client takes care of syncing and maintaining a local copy of the blockchain. It allows us to broadcast transactions and interact with deployed contracts. The two main clients we will use are geth and TestRPC. TestRPC provides a local dev chain, and geth allows us to connect to the mainnet and Rinkeby testnet. In the next chapter, we will use the tools and concepts from this chapter to broadcast a simple transaction and deploy our first contract.