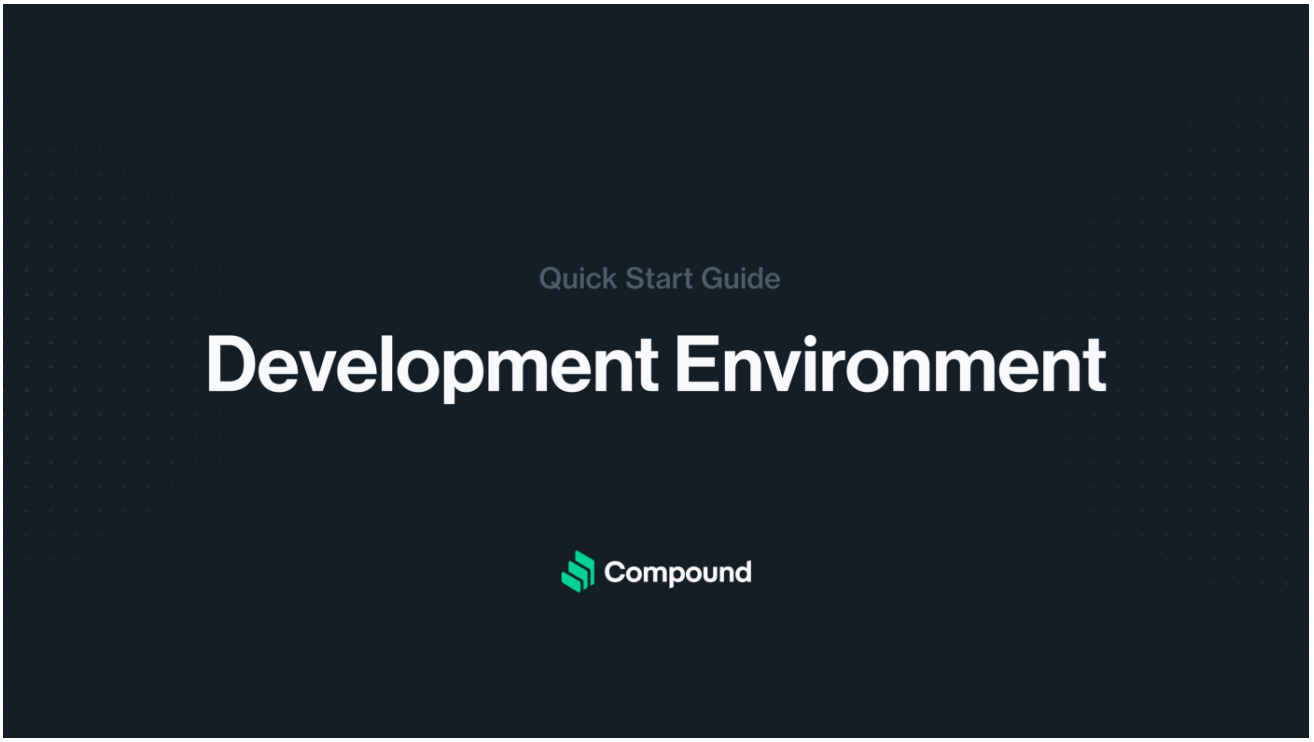
Setting up an Ethereum Development Environment

Quick Start Guide

Adam Bavosa Follow
Feb 10, 2020 · 10 min read



The Compound Protocol is a suite of Ethereum smart contracts that enable software developers to interface with cryptocurrency money markets. In order to supply or borrow assets from the protocol, you need to write to the Ethereum blockchain.

Ethereum is a public network of decentralized nodes that process transactions and append them to an ever-growing ledger, which is known as the blockchain. Developers can write software called smart contracts that are hosted on the Ethereum network. Anyone with access to Ethereum can invoke write and read operations to and from the blockchain.

In order to create a program for Ethereum and interact with Compound's smart

contracts, you need to have an access point to the network, some knowledge of smart contract architecture, and also some knowledge of JSON RPC.

Ethereum Development Key Concepts

Setting up your dev environment for Ethereum development takes just a few minutes. It's important to have a basic understanding of a few general concepts before we start writing code.

1. **Solidity** - The most popular programming language for writing smart contracts on Ethereum.
2. **Web3.js** - A JavaScript library for web browsers and Node.js that enables developers to elegantly read and write to the Ethereum blockchain. Smart contracts written in Solidity can be executed using JSON RPC from Web3.js.
3. **Infura.io** - A company that provides an API for simple Ethereum network access through HTTP and WebSockets. To access the network without a service like Infura's API, you must host your own Ethereum network node.
4. **Ethereum Networks** - The Ethereum Main Network (a.k.a Homestead) is considered to be the production environment. This is where real Ether (ETH) can be exchanged, burned, and mined. Also, there are popular public test networks: Ropsten, Göerli, Kovan, and Rinkeby. All of these networks can be accessed via Infura's API.
5. **Cloudflare's Ethereum Gateway** - Cloudflare, a popular DNS and general web-infrastructure company provides a free API for accessing the Ethereum Main Net. **Unlike Infura, Cloudflare's gateway can be used without an account or API**. The network can be interfaced with at https://cloudflare-eth.com. However, you can interface with only Homestead; not Ropsten, Göerli, Kovan, or Rinkeby.
6. **Truffle** - A company that creates Ethereum development tools like Truffle, Ganache, and Drizzle. In this guide, we'll be installing Ganache CLI so we can run our Solidity and Web3.js code locally during development.
7. **Ganache CLI** - A command line interface for running a locally hosted instance of Ethereum. We can start up a blank Ethereum blockchain, or a fork of a public network's blockchain on your local machine.
8. **Solc** - The Solidity compiler. This turns our human-readable Solidity code into Ethereum bytecode, which Ethereum network nodes understand how to execute.

9. **MetaMask** - A web browser extension that facilitates invocation of Ethereum smart contracts from a web page. MetaMask uses Web3.js under the hood. It is a tool for end users to use ETH, Ethereum based cryptocurrency tokens, and keep track of their Ethereum wallets when using decentralized, Web3 applications (a.k.a. DApps).

Installing MetaMask

MetaMask is a web browser extension that can be installed on Google Chrome, FireFox, Opera, or Brave. End users can use the extension to interface with smart contracts.

For example, when you visit https://app.compound.finance/, you can use MetaMask to supply your cryptocurrency assets to the Compound Protocol. The act of supplying assets on that website uses Web3.js and JSON RPC under the hood.

Those remote procedure calls are invoking Compound's smart contract functions, like mint, to accept your assets in exchange for cTokens.

You can set the network on MetaMask to point to any of the public networks, your own network, and even your localhost network. As you're developing your DApp's front-end web page, you can also test the back-end of the DApp locally if you are running Ganache CLI. This makes MetaMask a solid testing and development tool in addition to being an end user product.

The MetaMask browser extension **automatically imports the Web3 object as global JavaScript variable on every webpage** the user visits. It uses the HTTP provider in which is selected in the pictured drop down menu(more on providers later).

Installing the Solidity Compiler (solc)

Before you deploy a smart contract to an Ethereum network (or your localhost), you must first compile it to bytecode. This turns your program into machine-readable opcodes that Ethereum nodes understand how to execute.

In addition to bytecode, the Solidity compiler also produces an ABI (application binary interface). This is a JSON object that maps to your bytecode. Web3.js (and other frameworks) can do JSON RPC using the ABI, and the address of your smart contract. Think of the ABI as the bridge between Web2 and Web3.

Installing **solc** can be done using Homebrew on Mac or **apt-get** and the like on Linux distributions. Windows users need to build from source. Solidity compiler installation instructions for any OS can be found in the Solidity documentation.

Be sure to install the version of solc that corresponds to the `pragma` line in your smart contract code. The Solidity code we will write later uses **version 5**.

Once you have installed the Solidity compiler, you can use it on the command line to compile a Solidity file. If compilation succeeds, you'll get bytecode and a JSON ABI.

```
solc MySmartContracts.sol
```

Installing Web3.js

Web3.js is a popular library for interfacing with the Ethereum blockchain. To use it in a web page, you can import the library directly using a CDN like JSDeliver.

```
<script src="https://cdn.jsdelivr.net/npm/web3@latest
/dist/web3.min.js"></script>
```

To install the library for your built, front-end web project, or for a Node.js script/ server application, you can use NPM or Yarn for Node.js. Click here to install the LTS of Node.js and NPM.

```
npm install web3

## or

yarn add web3
```

To import Web3.js into a Node.js script or Browserify front-end project, you can use the following line of JavaScript:

```
const Web3 = require('web3');
```

If you imported via CDN directly to a web page, the `Web3` variable will already be declared; no need for the line of code above.

To initialize your Web3 object, you need to provide a network WebSocket or HTTP

provider in which to point Web3. This is the default HTTP provider address and port for Ganache CLI, which you can run on your local machine.

```
const web3 = new Web3('http://localhost:8545');
```

If you want to connect to a public network via Infura, you can do so once you create an Infura account and get your API key.

```
const web3 = new Web3('https://mainnet.infura.io/v3/_YOUR_API_KEY_');

// Infura can also access public test nets
// by changing the subdomain.
// Eg: https://ropsten.infura.io/v3/_YOUR_API_KEY_
```

Infura

Unless you already have your own Ethereum network nodes to be your entrypoint to the Ethereum network, you'll need to make an account with a service like Infura. You get up to 100,000 API calls per day, for free, if you make a developer account at https://infura.io/.

Create a project, and copy your "Endpoint." This contains your personal API key. Remember that you can change the subdomain of the URL to any of the public networks to switch the network your code interfaces with.

Ganache CLI

Ganache CLI, made by the Truffle team, is an essential tool for Ethereum development and testing. You can easily spin-up your own instance of Ethereum on your local machine with 1 terminal command. When you boot up Ganache, it gives you 10 Ethereum wallets that each contain 100 fake ETH. You can use this test net ETH to test smart contracts on your localhost blockchain.

```
ganache-cli
```

In addition to a blank-slate blockchain, you can also fork an existing public network. You would want to do a fork of a public net if you want to test out existing, public smart contracts on your local machine. Let's unpack an example.

Testing Out Compound's Smart Contracts

If you want to test out supplying Ethereum assets to Compound's Main network smart contracts, you can run Ganache with a fork of Main net. You can use the same addresses as the production Compound Protocol contracts, except interacting with them will only change the blockchain running on your local machine. Easy, streamlined development and testing.

```
ganache-cli -f https://cloudflare-eth.com
```

Installing Ganache CLI can be done with NPM or Yarn. If you haven't already click here to install the LTS of Node.js and NPM.

```
## Installs Ganache CLI globally so you can run it
## anywhere with `ganache-cli`

npm install -g ganache-cli

## or: yarn global add ganache-cli
```

Programming Your First DApp

Now that we have an understanding of basic Ethereum concepts, and we've set up our dev environment, we can write a full DApp. Total development time: **5 minutes**.

We're not going to use the Compound Protocol just yet. Baby steps. After you complete this walkthrough, you'll be ready to learn how to Supply Assets to the Compound Protocol, which is the next quick start guide.

Writing a Smart Contract with Solidity

Create a directory on your local machine for your DApp project files. You can do this on the command line.

```
mkdir first-ethereum-dapp

cd first-ethereum-dapp/
```

We'll write our smart contract in a Solidity file called FirstContract.sol.

```
touch FirstContract.sol

## or on non-bash Windows CLI: fsutil file createnew
FirstContract.sol 0
```

Open the Solidity file with your favorite text editor for writing code. I'll be using Sublime text, which has a community-made syntax highlighter for Solidity.

Here is the code for our contract file.

The contract has 1 getter function, and all it does is return 123. Once you write the code,

save this file.

Compiling a Solidity Smart Contract

Next we'll build our contract using the Solidity compiler CLI. On your command line, run the following.

```
solc --bin --abi -o ./build FirstContract.sol
```

This will create a folder called "build" in your project directory and also write 2 files to that folder. One contains the contract bytecode, and the other contains the contract ABI.

Configuring Your Project and Local Development Environment

Next, we'll need to use a Node.js script to deploy the code to Ganache. If you haven't already, install Node.js. Click here to install the LTS of Node.js and NPM.

From your project directory, run the following command on the command line.

```
npm init -y

## OR if you're a yarn person

## yarn init -y
```

This creates a package.json file which keeps track of project metadata and dependency data. There's no need to view or edit this file for now. Next, let's install our dependencies.

```
npm install web3 ganache-cli http-server

## alternatively
## yarn add web3 ganache-cli http-server
```

To run scripts inside of the `node_modules/.bin/` folder, you can globally install the `npx` module.

```
npm install -g npx
```

```
## or: yarn global add npx
```

Create a second terminal window and navigate to the project directory using `cd`. Here, we are going to run our instance of Ganache.

```
cd first-ethereum-dapp/
npx ganache-cli
```

Deploying to Your Local Test Net

Now that we have our own test blockchain running on our local machine, we can deploy the contract we built earlier. In your first command line window, make a new file called **deploy.js**.

```
touch deploy.js
```

Add the following code to the file. Running this script will deploy our smart contract to the instance of Ganache, which is running in our other command line window.

Save the file and run it using the following command.

```
node deploy.js
```

If successful, you will see the following output on your command line.

```
> FirstContract was successfully deployed!
> FirstContract can be interfaced with at this address:
> 0x702f935d608Aadf90323310c489B2903af20AA43
```

The hexadecimal number you see is the address on the blockchain where our contract now lives. Your address might be different from mine! **Copy this value and save it for later**.

Writing the Front-End Web Page of our DApp

The blockchain can be written to or read from with all modern web browsers, thanks to web3.js. Our DApp won't do any blockchain writes from the browser, so we won't be needing MetaMask for approving ETH transactions.

Create an HTML file in your project called index.html.

```
touch index.html
```

We'll put our user interface and our Web3.js code in this file.

Ethereum Ethereum Development Solidity Tutorial Web3

ject and point it to our local Ganache in the e call our smart contract function “getInteger” and update the UI with the result.

You might catch that we are missing something. We need the contract address, and also the ABI. Without these values, the Web3 object does not know where our contract lives or its callable methods.

Update the `myContractAddress` variable with the hexadecimal value you saw logged on the command line when you ran the deploy script. Make sure it is wrapped in quotes so it is of type string.

Copy the contents of `./build/FirstContract.abi` and paste it over the value of `myAbi` in the HTML file. Don't wrap this value in quotes. Now save the **index.html** file.

Running Your New Full-Stack DApp

We're ready to run our DApp! From the command line, run the following command to boot u a HTTP server, which will serve our HTML file when it is requested from a web browser.

```
npx http-server

> Starting up http-server, serving ./
> Available on:
> http://127.0.0.1:8080
> Hit CTRL-C to stop the server
```

As you can see from the command line log, we now have a server that can be reached at http://localhost:8080. Open up your favorite web browser and navigate to that address to access the DApp.

If everything works according to plan, you'll see something like the following.

When the web page loads, there is no integer value inside our HTML label. We are using web3.js to get the value from our smart contract function which is living on our local Ganach blockchain.

Congratulations! You have written your first full stack DApp. I hope this quick start guide go you up to speed on basic Ethereum DApp development. In the next quick start guide, we wil run through supplying Ethereum assets to the Compound Protocol from JSON RPC, and also

from our own Solidity smart contracts on the blockchain.

Thanks for reading and be sure to subscribe to the Compound Newsletter. Feel free to comment on this post, or get in touch in the #development room on our very active Discord server.